

The Portals 3.3 Message Passing Interface

Revision 1.0

Ron Brightwell¹, Arthur B. Maccabe², Rolf Riesen and Trammell Hudson

May 16, 2003

¹R. Brightwell and R. Riesen are with the Scalable Computing Systems Department, Sandia National Laboratories, P.O. Box 5800, Albuquerque, NM 87111-1110, bright@cs.sandia.gov, rolf@cs.sandia.gov.

²A. B. Maccabe is with the Computer Science Department, University of New Mexico, Albuquerque, NM 87131-1386, maccabe@cs.unm.edu.

Abstract

This report presents a specification for the Portals 3.3 message passing interface. Portals 3.3 is intended to allow scalable, high-performance network communication between nodes of a parallel computing system. Specifically, it is designed to support a parallel computing platform composed of clusters of commodity workstations connected by a commodity system area network fabric. In addition, Portals 3.3 is well suited to massively parallel processing and embedded systems. Portals 3.3 represents an adaption of the data movement layer developed for massively parallel processing platforms, such as the 4500-node Intel TeraFLOPS machine.

Contents

1	Introduction	1
1.1	Overview	1
1.2	Purpose	1
1.3	Background	2
1.4	Scalability	2
1.5	Communication Model	2
1.6	Zero Copy, OS Bypass and Application Bypass	3
1.7	Faults	3
2	An Overview of the Portals API	4
2.1	Data Movement	4
2.2	Portal Addressing	5
2.3	Access Control	6
2.4	Multi-threaded Applications	7
3	The Portals API	8
3.1	Naming Conventions	8
3.2	Base Types	8
3.2.1	Sizes	8
3.2.2	Handles	8
3.2.3	Indexes	9
3.2.4	Match Bits	9
3.2.5	Network Interfaces	9
3.2.6	Identifiers	9
3.2.7	Status Registers	9
3.3	Return Codes	9
3.4	Initialization and Cleanup	10
3.4.1	PtlInit	10
3.4.2	PtlFini	10
3.5	Network Interfaces	10
3.5.1	The Network Interface Limits Type	11
3.5.2	PtlNIInit	11
3.5.3	PtlNIFini	12
3.5.4	PtlNIStatus	13
3.5.5	PtlNIDist	13
3.5.6	PtlNIHandle	14
3.6	User Identification	14
3.6.1	PtlGetUid	14
3.7	Process Identification	15
3.7.1	The Process Id Type	15
3.7.2	PtlGetId	15
3.8	Process Aggregation	15

3.8.1	The Job Id Type	16
3.9	Match List Entries and Match Lists	16
3.9.1	Match Entry Type Definitions	16
3.9.2	PtlMEAttach	16
3.9.3	PtlMEAttachAny	17
3.9.4	PtlMEInsert	18
3.9.5	PtlMEUnlink	19
3.10	Memory Descriptors	19
3.10.1	The Memory Descriptor Type	19
3.10.2	The Memory Descriptor IO Vector Type	21
3.10.3	PtlMDAttach	21
3.10.4	PtlMDBind	22
3.10.5	PtlMDUnlink	22
3.10.6	PtlMDUpdate	23
3.10.7	Thresholds and Unlinking	24
3.11	Events and Event Queues	24
3.11.1	Kinds of Events	24
3.11.2	Event Ordering	25
3.11.3	Failure Notification	25
3.11.4	The Event Type	25
3.11.5	The Event Queue Handler Type	26
3.11.6	PtlEQAlloc	27
3.11.7	Event Queue Handler Semantics	27
3.11.8	PtlEQFree	28
3.11.9	PtlEQGet	28
3.11.10	PtlEQWait	29
3.11.11	PtlEQPoll	29
3.11.12	Event Semantics	30
3.12	The Access Control Table	30
3.12.1	PtlACEntry	30
3.13	Data Movement Operations	31
3.13.1	Portal Acknowledgment Type Definition	31
3.13.2	PtlPut	31
3.13.3	PtlPutRegion	32
3.13.4	PtlGet	33
3.13.5	PtlGetRegion	33
3.13.6	PtlGetPut	34
3.14	PtlHandleIsEqual	35
3.15	Summary	35
4	The Semantics of Message Transmission	40
4.1	Sending Messages	40
4.2	Receiving Messages	41

List of Figures

2.1	Portal Put (Send)	4
2.2	Portal Get	5
2.3	Portal Swap (Getput)	5
2.4	Portal Addressing Structures	6
2.5	Portals Address Translation	6

List of Tables

3.1	Object Type Codes	8
3.2	Types Defined by the Portals 3.3 API	36
3.3	Functions Defined by the Portals 3.3 API	37
3.4	Function Return Codes for the Portals 3.3 API	38
3.5	Other Constants Defined by the Portals 3.3 API	39
4.1	Information Passed in a Put Request	40
4.2	Information Passed in an Acknowledgement	41
4.3	Information Passed in a Get Request	41
4.4	Information Passed in a Reply	42

Glossary

API Application Programming Interface. A definition of the functions and semantics provided by library of functions.

Initiator A *process* that initiates a message operation.

Message An application-defined unit of data that is exchanged between *processes*.

Message Operation Either a put operation, which writes data, or a get operation, which reads data.

Network A network provides point-to-point communication between *nodes*. Internally, a network may provide multiple routes between endpoints (to improve fault tolerance or to improve performance characteristics); however, multiple paths will not be exposed outside of the network.

Node A node is an endpoint in a *network*. Nodes provide processing capabilities and memory. A node may provide multiple processors (an SMP node) or it may act as a *gateway* between networks.

Process A context of execution. A process defines a virtual memory (VM) context. This context is not shared with other processes. Several threads may share the VM context defined by a process.

Target A *process* that is acted upon by a message operation.

Thread A context of execution that shares a VM context with other threads.

Chapter 1

Introduction

1.1 Overview

This document describes an application programming interface for message passing between nodes in a system area network. The goal of this interface is to improve the scalability and performance of network communication by defining the functions and semantics of message passing required for scaling a parallel computing system to ten thousand nodes. This goal is achieved by providing an interface that will allow a quality implementation to take advantage of the inherently scalable design of Portals.

This document is divided into several sections:

Section 1—Introduction This section describes the purpose and scope of the Portals API.

Section 2—An Overview of the Portals 3.1 API This section gives a brief overview of the Portals API. The goal is to introduce the key concepts and terminology used in the description of the API.

Section 3—The Portals 3.3 API This section describes the functions and semantics of the Portals application programming interface.

Section 4—The Semantics of Message Transmission This section describes the semantics of message transmission. In particular, the information transmitted in each type of message and the processing of incoming messages.

1.2 Purpose

Existing message passing technologies available for commodity cluster networking hardware do not meet the scalability goals required by the Cplant [1] project at Sandia National Laboratories. The goal of the Cplant project is to construct a commodity cluster that can scale to the order of ten thousand nodes. This number greatly exceeds the capacity for which existing message passing technologies have been designed and implemented.

In addition to the scalability requirements of the network, these technologies must also be able to support a scalable implementation of the Message Passing Interface (MPI) [7] standard, which has become the *de facto* standard for parallel scientific computing. While MPI does not impose any scalability limitations, existing message passing technologies do not provide the functionality needed to allow implementations of MPI to meet the scalability requirements of Cplant.

The following are properties of a network architecture that do not impose any inherent scalability limitations:

- **Connectionless** - Many connection-oriented architectures, such as VIA [3] and TCP/IP sockets, have limitations on the number of peer connections that can be established.
- **Network independence** - Many communication systems depend on the host processor to perform operations in order for messages in the network to be consumed. Message consumption from the network should not be dependent on host processor activity, such as the operating system scheduler or user-level thread scheduler.

- User-level flow control - Many communication systems manage flow control internally to avoid depleting resources, which can significantly impact performance as the number of communicating processes increases.
- OS Bypass - High performance network communication should not involve memory copies into or out of a kernel-managed protocol stack.

The following are properties of a network architecture that do not impose scalability limitations for an implementation of MPI:

- Receiver-managed - Sender-managed message passing implementations require a persistent block of memory to be available for every process, requiring memory resources to increase with job size and requiring user-level flow control mechanisms to manage these resources.
- User-level Bypass - While OS Bypass is necessary for high-performance, it alone is not sufficient to support the Progress Rule of MPI asynchronous operations.
- Unexpected messages - Few communication systems have support for receiving messages for which there is no prior notification. Support for these types of messages is necessary to avoid flow control and protocol overhead.

1.3 Background

Portals was originally designed for and implemented on the nCube machine as part of the SUNMOS (Sandia/UNM OS) [6] and Puma [11] lightweight kernel development projects. Portals went through two design phases, the latter of which is used on the 4500-node Intel TeraFLOPS machine [10]. Portals have been very successful in meeting the needs of such a large machine, not only as a layer for a high-performance MPI implementation [2], but also for implementing the scalable run-time environment and parallel I/O capabilities of the machine.

The second generation Portals implementation was designed to take full advantage of the hardware architecture of large MPP machines. However, efforts to implement this same design on commodity cluster technology identified several limitations, due to the differences in network hardware as well as to shortcomings in the design of Portals.

1.4 Scalability

The primary goal in the design of Portals is scalability. Portals are designed specifically for an implementation capable of supporting a parallel job running on tens of thousands of nodes. Performance is critical only in terms of scalability. That is, the level of message passing performance is characterized by how far it allows an application to scale and not by how it performs in micro-benchmarks (e.g., a two node bandwidth or latency test).

The Portals API is designed to allow for scalability, not to guarantee it. Portals cannot overcome the shortcomings of a poorly designed application program. Applications that have inherent scalability limitations, either through design or implementation, will not be transformed by Portals into scalable applications. Scalability must be addressed at all levels. Portals do not inhibit scalability, but do not guarantee it either.

To support scalability, the Portals interface maintains a minimal amount of state. Portals provide reliable, ordered delivery of messages between pairs of processes. They are connectionless: a process is not required to explicitly establish a point-to-point connection with another process in order to communicate. Moreover, all buffers used in the transmission of messages are maintained in user space. The target process determines how to respond to incoming messages, and messages for which there are no buffers are discarded.

1.5 Communication Model

Portals combine the characteristics of both one-side and two-sided communication. They define a “matching put” operation and a “matching get” operation. The destination of a put (or send) is not an explicit address; instead, each message contains a set of match bits that allow the receiver to determine where incoming messages should be placed. This flexibility allows Portals to support both traditional one-sided operations and two-sided send/receive operations.

Portals allows the target to determine whether incoming messages are acceptable. A target process can choose to accept message operations from any specific process or can choose to ignore message operations from any specific process.

1.6 Zero Copy, OS Bypass and Application Bypass

In traditional system architectures, network packets arrive at the network interface card (NIC), are passed through one or more protocol layers in the operating system, and eventually copied into the address space of the application. As network bandwidth began to approach memory copy rates, reduction of memory copies became a critical concern. This concern led to the development of zero-copy message passing protocols in which message copies are eliminated or pipelined to avoid the loss of bandwidth.

A typical zero-copy protocol has the NIC generate an interrupt for the CPU when a message arrives from the network. The interrupt handler then controls the transfer of the incoming message into the address space of the appropriate application. The interrupt latency, the time from the initiation of an interrupt until the interrupt handler is running, is fairly significant. To avoid this cost, some modern NICs have processors that can be programmed to implement part of a message passing protocol. Given a properly designed protocol, it is possible to program the NIC to control the transfer of incoming messages, without needing to interrupt the CPU. Because this strategy does not need to involve the OS on every message transfer, it is frequently called “OS Bypass.” ST [12], VIA [3], FM [5], GM [9], and Portals are examples of OS Bypass protocols.

Many protocols that support OS Bypass still require that the application actively participate in the protocol to ensure progress. As an example, the long message protocol of PM requires that the application receive and reply to a request to put or get a long message. This complicates the runtime environment, requiring a thread to process incoming requests, and significantly increases the latency required to initiate a long message protocol. The Portals message passing protocol does not require activity on the part of the application to ensure progress. We use the term “Application Bypass” to refer to this aspect of the Portals protocol.

1.7 Faults

Given the number of components that we are dealing with and the fact that we are interested in supporting applications that run for very long times, failures are inevitable. The Portals API recognizes that the underlying transport may not be able to successfully complete an operation once it has been initiated. This is reflected in the fact that the Portals API reports three types of events: events indicating the initiation of an operation, events indicating the successful completion of an operation, and events indicating the unsuccessful completion of an operation. Every initiation event is eventually followed by a successful completion event or an unsuccessful completion event.

Between the time an operation is started and the time that the operation completes (successfully or unsuccessfully), any memory associated with the operation should be considered volatile. That is, the memory may be changed in unpredictable ways while the operation is progressing. Once the operation completes, the memory associated with the operation will not be subject to further modification (from this operation). Notice that unsuccessful operations may alter memory in an essentially unpredictable fashion.

Chapter 2

An Overview of the Portals API

In this section, we give a conceptual overview of the Portals API. The goal is to provide a context for understanding the detailed description of the API presented in the next section.

2.1 Data Movement

A Portal represents an opening in the address space of a process. Other processes can use a Portal to read (get), write (put), or atomically swap (getput) the memory associated with the portal. Every data movement operation involves two processes, the **initiator** and the **target**. The initiator is the process that initiates the data movement operation. The target is the process that responds to the operation by either accepting the data for a put operation, replying with the data for a get operation, or both for a swap operation.

In this discussion, activities attributed to a process may refer to activities that are actually performed by the process or *on behalf of the process*. The inclusiveness of our terminology is important in the context of *application bypass*. In particular, when we note that the target sends a reply in the case of a get operation, it is possible that reply will be generated by another component in the system, bypassing the application.

Figures 2.1, 2.2, 2.3 present graphical interpretations of the Portal data movement operations: put, get, and swap. In the case of a put operation, the initiator sends a put request message containing the data to the target. The target translates the Portal addressing information in the request using its local Portal structures. When the request has been processed, the target optionally sends an acknowledgement message.

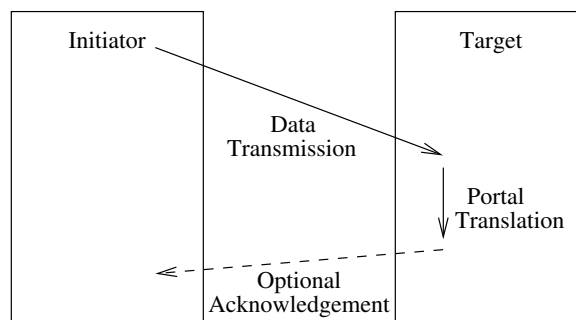


Figure 2.1: Portal Put (Send)

In the case of a get operation, the initiator sends a get request to the target. As with the put operation, the target translates the Portal addressing information in the request using its local Portal structures. Once it has translated the Portal addressing information, the target sends a reply that includes the requested data.

We should note that Portal address translations are only performed on nodes that respond to operations initiated by other nodes. Acknowledgements and replies to get operations bypass the portals address translation structures.

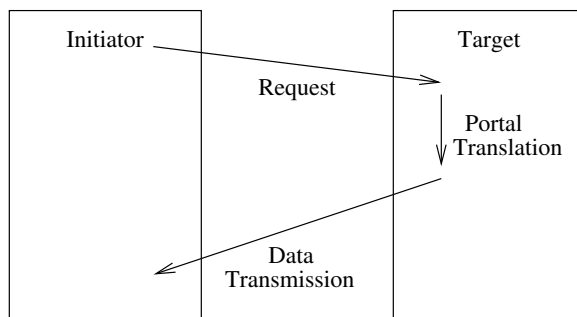


Figure 2.2: Portal Get

In the case of a swap operation, the initiator sends a getput message containing data to the target. The target translates the Portal addressing information using its local Portal structures. The target then sends the current data in a reply message and deposits the new data from the initiator.

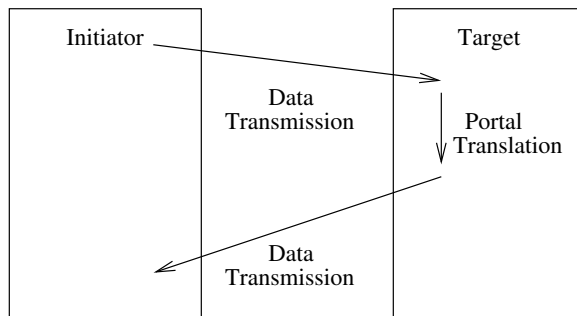


Figure 2.3: Portal Swap (Getput)

2.2 Portal Addressing

One-sided data movement models (e.g., shmem [4], ST [12], MPI-2 [8]) typically use a triple to address memory on a remote node. This triple consists of a process id, memory buffer id, and offset. The process id identifies the target process, the memory buffer id specifies the region of memory to be used for the operation, and the offset specifies an offset within the memory buffer.

In addition to the standard address components (process id, memory buffer id, and offset), a Portal address includes a set of match bits. This addressing model is appropriate for supporting one-sided operations as well as traditional two-sided message passing operations. Specifically, the Portals API provides the flexibility needed for an efficient implementation of MPI-1, which defines two-sided operations with one-sided completion semantics.

Figure 2.4 presents a graphical representation of the structures used by a target in the interpretation of a Portal address. The process id is used to route the message to the appropriate node and is not reflected in this diagram. The memory buffer id, called the **portal id**, is used as an index into the Portal table. Each element of the Portal table identifies a match list. Each element of the match list specifies two bit patterns: a set of “don’t care” bits, and a set of “must match” bits. In addition to the two sets of match bits, each match list element has at most one memory descriptor. Each memory descriptor identifies a memory region and an optional event queue. The memory region specifies the memory to be used in the operation and the event queue is used to record information about these operations.

Figure 2.5 illustrates the steps involved in translating a Portal address, starting from the first element in a match list.

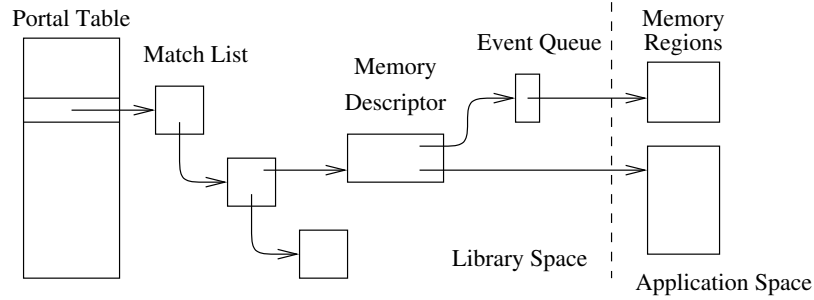


Figure 2.4: Portal Addressing Structures

If the match criteria specified in the match list entry are met and the memory descriptor list accepts the operation¹, the operation (put, get, or swap) is performed using the memory region specified in the memory descriptor. If the memory descriptor specifies that it is to be unlinked when a threshold has been exceeded, the match list entry is removed from the match list and the resources associated with the memory descriptor and match list entry are reclaimed. Finally, if there is an event queue specified in the memory descriptor and the memory descriptor accepts the event, the operation is logged in the event queue.

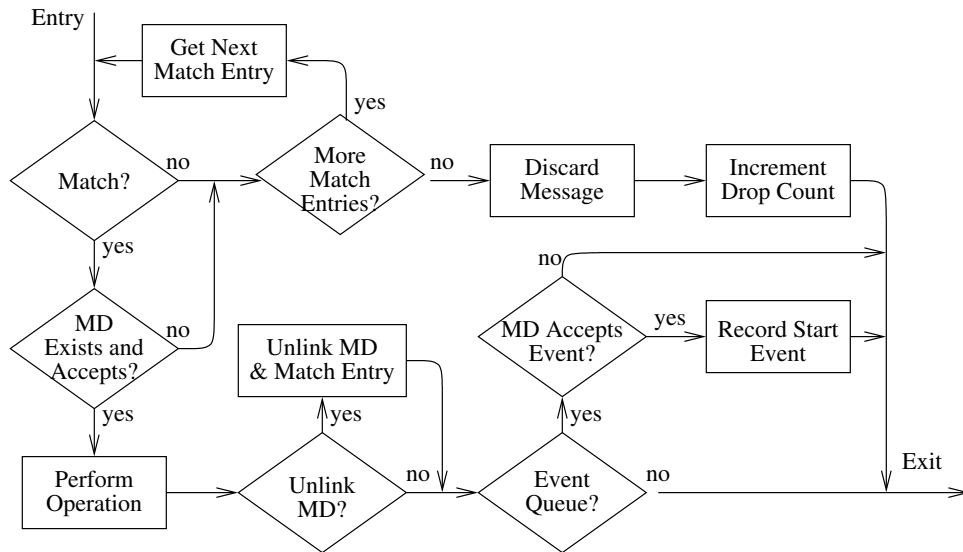


Figure 2.5: Portals Address Translation

If the match criteria specified in the match list entry are not met, or there is no memory descriptor associated with the match list entry, or the memory descriptor associated with the match list entry rejects the operation, the address translation continues with the next match list entry. If the end of the match list has been reached, the address translation is aborted and the incoming requested is discarded.

2.3 Access Control

A process can control access to its Portals using an access control list. Each entry in the access control list specifies a process id, possibly a job id, a user id, and a Portal table index. The access control list is actually an array of entries.

¹Memory descriptors can reject operations because a threshold has been exceeded or because the memory region does not have sufficient space, see Section 3.10

Each incoming request includes an index into the access control list (i.e., a “cookie” or hint). If the id of the process issuing the request doesn’t match the id specified in the access control list entry or the Portal table index specified in the request doesn’t match the Portal table index specified in the access control list entry, the request is rejected. Process identifiers, job identifiers, user identifiers, and Portal table indexes may include wildcard values to increase the flexibility of this mechanism.

Two aspects of this design merit further discussion. First, the model assumes that the information in a message header, the sender’s process id, node id, user id, and job id in particular, is trustworthy. In most contexts, we assume that the entity that constructs the header is trustworthy; however, using cryptographic techniques, we could easily devise a protocol that would ensure the authenticity of the sender.

Second, because the access check is performed by the receiver, it is possible that a malicious process will generate thousands of messages that will be denied by the receiver. This could saturate the network and/or the receiver, resulting in a *denial of service* attack. Moving the check to the sender using capabilities, would remove the potential for this form of attack. However, the solution introduces the complexities of capability management (exchange of capabilities, revocation, protections, etc).

2.4 Multi-threaded Applications

The Portals API supports a generic view of multi-threaded applications. From the perspective of the Portals API, an application program is defined by a set of processes. Each process defines a unique address space. The Portals API defines access to this address space from other processes (using Portals addressing and the data movement operations). A process may have one or more *threads* executing in its address space.

With the exception of *PtEQWait* and possibly *PtEQPoll*, every function in the Portals API is non-blocking and atomic with respect to both other threads and external operations that result from data movement operations. While individual operations are atomic, sequences of these operations may be interleaved between different threads and with external operations. The Portals API does not provide any mechanisms to control this interleaving. It is expected that these mechanisms will be provided by the API used to create threads.

Chapter 3

The Portals API

3.1 Naming Conventions

The Portals API defines two types of entities: functions and types. Function always start with *Ptl* and use mixed upper and lower case. When used in the body of this report, function names appear in italic face, e.g., *PtlInit*. The functions associated with an object type will have names that start with *Ptl*, followed by the two letter object type code shown in Table 3.1. As an example, the function *PtlEQAlloc* allocates resources for an event queue.

Table 3.1: Object Type Codes

<i>xx</i>	Name	Section
eq	Event Queue	3.11
md	Memory Descriptor	3.10
me	Match list Entry	3.9
ni	Network Interface	3.5

Type names use lower case with underscores to separate words. Each type name starts with `ptl_` and ends with `_t`. When used in the body of this report, type names appear in a fixed font, e.g., `ptl_match_bits_t`.

Names for constants use upper case with underscores to separate words. Each constant name starts with `PTL_`. When used in the body of this report, type names appear in a fixed font, e.g., `PTL_OK`.

The definition of named constants, function prototypes, and type definitions must be supplied in an include file named `portals3.h`.

3.2 Base Types

The Portals API defines a variety of base types. These types represent a simple renaming of the base types provided by the C programming language. In most cases these new type names have been introduced to improve type safety and to avoid issues arising from differences in representation sizes (e.g., 16-bit or 32-bit integers).

3.2.1 Sizes

The type `ptl_size_t` is an unsigned 64-bit integral type used for representing sizes.

3.2.2 Handles

Objects maintained by the API are accessed through handles. Handle types have names of the form `ptl_handle_xx_t`, where *xx* is one of the two letter object type codes shown in Table 3.1. For example, the type `ptl_handle_ni_t` is used for network interface handles.

Each type of object is given a unique handle type to enhance type checking. The type, `ptl_handle_any_t`, can be used when a generic handle is needed. Every handle value can be converted into a value of type `ptl_handle_any_t` without loss of information.

Handles are not simple values. Every portals object is associated with a specific network interface and an identifier for this interface (along with an object identifier) is part of the handle for the object.

The special value `PTL_EQ_NONE`, of type `ptl_handle_eq_t`, is used to indicate the absence of an event queue. See sections 3.10.5 and 3.10.6 for uses of this value. The special value `PTL_INVALID_HANDLE` is used to represent an invalid handle.

3.2.3 Indexes

The types `ptl_pt_index_t` and `ptl_ac_index_t` are integral types used for representing Portal table indexes and access control table indexes respectively. See section 3.5.2 for limits on values of these types.

3.2.4 Match Bits

The type `ptl_match_bits_t` is capable of holding unsigned 64-bit integer values.

3.2.5 Network Interfaces

The type `ptl_interface_t` is an integral type used for identifying different network interfaces. Users will need to consult the implementation documentation to determine appropriate values for the interfaces available. The special value `PTL_IFACE_DEFAULT` identifies the default interface.

3.2.6 Identifiers

The type `ptl_nid_t` is an integral type used for representing node ids, `ptl_pid_t` is an integral type for representing process ids, `ptl_uid_t` is an integral type for representing user ids, and `ptl_jid_t` is an integral type for representing job ids.

The special values `PTL_PID_ANY` matches any process identifier, `PTL_NID_ANY` matches any node identifier, `PTL_UID_ANY` matches any user identifier, and `PTL_JID_ANY` matches any job identifier. See sections 3.9.2 and 3.12.1 for uses of these values.

3.2.7 Status Registers

Each network interface maintains an array of status registers that can be accessed using the `ptlNIStatus` function (see Section 3.5.4). The type `ptl_sr_index_t` defines the types of indexes that can be used to access the status registers. The only index defined for all implementations is `PTL_SR_DROP_COUNT` which identifies the status register that counts the dropped requests for the interface. Other indexes (and registers) may be defined by the implementation.

The type `ptl_sr_value_t` defines the types of values held in status registers. This is a signed integer type. The size is implementation dependent but must be at least 32 bits.

3.3 Return Codes

The API specifies return codes that indicate success or failure of a function call. In the case where the failure is due to invalid arguments being passed into the function, the exact behavior of an implementation is undefined. The API suggests error codes that provide more detail about specific invalid parameters, but an implementation is not required to return these specific error codes. For example, an implementation is free to allow the caller to fault when given an invalid address, rather than return `PTL_SEGV`. In addition, an implementation is free to map these return codes to standard return codes where appropriate. For example, a Linux kernel-space implementation may want to map Portals return codes to POSIX-compliant return codes.

3.4 Initialization and Cleanup

The Portals API includes a function, *PtlInit*, to initialize the library and a function, *PtlFini*, to cleanup after the process is done using the library.

A child process does not inherit any Portals resources from its parent. A child process whose parent has initialized Portals must shutdown and re-initialize Portals in order to obtain new, valid, Portals resources. If a child process fails to shutdown and re-initialize Portals, behavior is undefined for both the parent and the child.

3.4.1 PtlInit

```
int PtlInit( int *max_interfaces );
```

The *PtlInit* function initializes the Portals library. *PtlInit* must be called at least once by a process before any thread makes a Portals function call but may be safely called more than once.

Return Codes

PTL_OK Indicates success.

PTL_FAIL Indicates an error during initialization.

PTL_SEGV Indicates that `max_interfaces` is not a legal address.

Arguments

`max_interfaces` **output** On successful return, this location will hold the maximum number of interfaces that can be initialized.

3.4.2 PtlFini

```
void PtlFini( void );
```

The *PtlFini* function cleans up after the Portals library is no longer needed by a process. After this function is called, calls to any of the functions defined by the Portal API or use of the structures set up by the Portals API will result in undefined behavior. This function should be called once and only once during termination by a process. Typically, this function will be called in the exit sequence of a process. Individual threads should not call *PtlFini* when they terminate.

3.5 Network Interfaces

The Portals API supports the use of multiple network interfaces. However, each interface is treated as an independent entity. Combining interfaces (e.g., “bonding” to create a higher bandwidth connection) must be implemented by the process or embedded in the underlying network. Interfaces are treated as independent entities to make it easier to cache information on individual network interface cards.

Once initialized, each interface provides a Portal table, an access control table, and a collection of status registers. In order to facilitate the development of portable Portals applications, a compliant implementation must provide at least eight Portal table entries. See Section 3.9 for a discussion of updating Portal table entries using the *PtlMEAttach* or *PtlMEAttachAny* functions. See Section 3.12 for a discussion of the initialization and updating of entries in the access control table. See Section 3.5.4 for a discussion of the *PtlNISstatus* function which can be used to determine the value of a status register.

Every other type of Portals object (e.g., memory descriptor, event queue, or match list entry) is associated with a specific network interface. The association to a network interface is established when the object is created and is encoded in the handle for the object.

Each network interface is initialized and shutdown independently. The initialization routine, *PtlNIInit*, returns a handle for an interface object which is used in all subsequent Portals operations. The *PtlNIFini* function is used to

shutdown an interface and release any resources that are associated with the interface. Network interface handles are associated with processes, not threads. All threads in a process share all of the network interface handles.

The Portals API also defines the *PtlNIStatus* function to query the status registers for a network interface, the *PtlNIDist* function to determine the “distance” to another process, and the *PtlNIHandle* function to determine the network interface with which an object is associated.

3.5.1 The Network Interface Limits Type

```
typedef struct {
    int      max_mes;
    int      max_mds;
    int      max_eqs;
    int      max_ac_index;
    int      max_pt_index;
    int      max_md_iovecs;
    int      max_me_list;
    int      max_getput_md;
} ptl_ni_limits_t;
```

Values of type `ptl_ni_limits_t` include the following members:

max_mes Maximum number of match entries that can be allocated at any one time.

max_mds Maximum number of memory descriptors that can be allocated at any one time.

max_eqs Maximum number of event queues that can be allocated at any one time.

max_ac_index Largest access control table index for this interface, valid indexes range from zero to `max_ac_index`, inclusive.

max_pt_index Largest Portal table index for this interface, valid indexes range from zero to `max_pt_index`, inclusive.

max_md_iovecs Maximum number of io vectors for a single memory descriptor for this interface.

max_me_list Maximum number of match entries that can be attached to any Portal table index.

max_getput_md Maximum length of the local and remote memory descriptors used in the atomic swap *PtlGetPut* operation.

3.5.2 PtlNIInit

```
int PtlNIInit( ptl_interface_t  interface
               ptl_pid_t        pid,
               ptl_ni_limits_t* desired,
               ptl_ni_limits_t* actual,
               ptl_handle_ni_t* ni_handle );
```

The *PtlNIInit* function is used to initialize the Portals API for a network interface. This function must be called at least once by a process before any other operations that apply to the interface by any process or thread. For subsequent calls to *PtlNIInit* from within the same process (either by different threads or the same thread), the desired limits will be ignored and the call will return the existing NI handle.

Return Codes

PTL_OK Indicates success.

PTL_NO_INIT Indicates that the Portals API has not been successfully initialized.

PTL_IFACE_DUP Indicates a duplicate initialization of `interface`.

PTL_IFACE_INVALID Indicates that `interface` is not a valid network interface.

PTL_NO_SPACE Indicates that there is insufficient memory to initialize the interface.

PTL_PID_INVALID Indicates that `pid` is not a valid process id.

PTL_SEGV Indicates that `actual` or `ni_handle` is not a legal address.

Arguments

<code>interface</code>	input	Identifies the network interface to be initialized. (See section 3.2.5 for a discussion of values used to identify network interfaces.)
<code>pid</code>	input	Identifies the desired process id (for well known process ids). The value <code>PTL_PID_ANY</code> may be used to have the process id assigned by the underlying library.
<code>desired</code>	input	If non-NULL, points to a structure that holds the desired limits.
<code>actual</code>	output	if non-NULL, on successful return, the location pointed to by <code>actual</code> will hold the actual limits.
<code>ni_handle</code>	output	On successful return, this location will hold a handle for the interface.

Discussion: *The use of `desired` is implementation dependent. In particular, an implementation may choose to ignore this argument*

The desired limits are used to offer a hint to an implementation as to the amount of resources needed, and the implementation returns the actual limits available for use. In the case where an implementation does not have any pre-defined limits, it is free to return the largest possible value permitted by the corresponding type (eg. `INT_MAX`). A quality implementation will enforce the limits that are returned and take the appropriate action when limits are exceeded, such as using the `PTL_NO_SPACE` return code. The caller is permitted to use maximum values for the desired fields to indicate that the limit should be determined by the implementation.

3.5.3 PtlNIFini

```
int PtlNIFini( ptl_handle_ni_t ni_handle );
```

The *PtlNIFini* function is used to release the resources allocated for a network interface. Once the *PtlNIFini* operation has been started, the results of pending API operations (e.g., operations initiated by another thread) for this interface are undefined. Similarly, the effects of incoming operations (puts and gets) or return values (acknowledgements and replies) for this interface are undefined.

Return Codes

PTL_OK Indicates success.

PTL_NO_INIT Indicates that the Portals API has not been successfully initialized.

PTL_NI_INVALID Indicates that `ni_handle` is not a valid network interface handle.

Arguments

<code>interface</code>	input	A handle for the interface to shutdown.
------------------------	--------------	---

3.5.4 PtlNIStatus

```
int PtlNIStatus( ptl_handle_ni_t ni_handle,
                 ptl_sr_index_t  status_register,
                 ptl_sr_value_t* status );
```

The *PtlNIStatus* function returns the value of a status register for the specified interface. (See section 3.2.7 for more information on status register indexes and status register values.)

Return Codes

PTL_OK Indicates success.

PTL_NO_INIT Indicates that the Portals API has not been successfully initialized.

PTL_NI_INVALID Indicates that *ni_handle* is not a valid network interface handle.

PTL_SR_INDEX_INVALID Indicates that *status_register* is not a valid status register.

PTL_SEGV Indicates that *status* is not a legal address.

Arguments

<i>ni_handle</i>	input	A handle for the interface to use.
<i>status_register</i>	input	An index for the status register to read.
<i>status</i>	output	On successful return, this location will hold the current value of the status register.

Discussion: *The only status register that must be defined is a drop count register (PTL_SR_DROP_COUNT). Implementations may define additional status registers. Identifiers for the indexes associated with these registers should start with the prefix PTL_SR_.*

3.5.5 PtlNIDist

```
int PtlNIDist( ptl_handle_ni_t  ni_handle,
               ptl_process_id_t process,
               unsigned long*    distance );
```

The *PtlNIDist* function returns the distance to another process using the specified interface. Distances are only defined relative to an interface. Distance comparisons between different interfaces on the same process may be meaningless.

Return Codes

PTL_OK Indicates success.

PTL_NO_INIT Indicates that the Portals API has not been successfully initialized.

PTL_NI_INVALID Indicates that *ni_handle* is not a valid network interface handle.

PTL_PROCESS_INVALID Indicates that *process* is not a valid process identifier.

PTL_SEGV Indicates that *distance* is not a legal address.

Arguments

<i>ni_handle</i>	input	A handle for the interface to use.
<i>process</i>	input	An identifier for the process whose distance is being requested.
<i>distance</i>	output	On successful return, this location will hold the distance to the remote process.

Discussion: *This function should return a static measure of distance. Examples include minimum latency, the inverse of available bandwidth, or the number of switches between the two endpoints.*

3.5.6 PtlNIHandle

```
int PtlNIHandle( ptl_handle_any_t handle,
                 ptl_handle_ni_t* ni_handle );
```

The *PtlNIHandle* function returns a handle for the network interface with which the object identified by `handle` is associated. If the object identified by `handle` is a network interface, this function returns the same value it is passed.

Return Codes

PTL_OK Indicates success.

PTL_NO_INIT Indicates that the Portals API has not been successfully initialized.

PTL_HANDLE_INVALID Indicates that `handle` is not a valid handle.

PTL_SEGV Indicates that `ni_handle` is not a legal address.

Arguments

<code>handle</code>	input	A handle for the object.
<code>ni_handle</code>	output	On successful return, this location will hold a handle for the network interface associated with <code>handle</code> .

Discussion: Every handle should encode the network interface and the object id relative to this handle. Both are presumably encoded using integer values.

3.6 User Identification

Every process runs on behalf of a user.

3.6.1 PtlGetUid

```
int PtlGetUid( ptl_handle_ni_t    ni_handle,
                ptl_uid_t*        uid );
```

Return Codes

PTL_OK Indicates success.

PTL_NI_INVALID Indicates that `ni_handle` is not a valid network interface handle.

PTL_NO_INIT Indicates that the Portals API has not been successfully initialized.

PTL_SEGV Indicates that `uid` is not a legal address.

Arguments

<code>ni_handle</code>	input	A network interface handle.
<code>uid</code>	output	On successful return, this location will hold the user id for the calling process.

Discussion: Note that user identifiers are dependent on the network interface(s). In particular, if a node has multiple interfaces, a process may have multiple user identifiers.

3.7 Process Identification

Processes that use the Portals API, can be identified using a node id and process id. Every node accessible through a network interface has a unique node identifier and every process running on a node has a unique process identifier. As such, any process in the computing system can be uniquely identified by its node id and process id.

The Portals API defines a type, `ptl_process_id_t` for representing process ids, and a function, *PtlGetId*, which can be used to obtain the id of the current process.

Discussion: *The portals API does not include thread identifiers. Messages are delivered to processes (address spaces) not threads (contexts of execution).*

3.7.1 The Process Id Type

```
typedef struct {
    ptl_nid_t      nid;
    ptl_pid_t      pid;
} ptl_process_id_t;
```

The `ptl_process_id_t` type uses two identifiers to represent a process id: a node id and a process id.

3.7.2 PtlGetId

```
int PtlGetId( ptl_handle_ni_t  ni_handle,
              ptl_process_id_t* id );
```

Return Codes

PTL_OK Indicates success.

PTL_NI_INVALID Indicates that `ni_handle` is not a valid network interface handle.

PTL_NO_INIT Indicates that the Portals API has not been successfully initialized.

PTL_SEGV Indicates that `id` is not a legal address.

Arguments

<code>ni_handle</code>	input	A network interface handle.
<code>id</code>	output	On successful return, this location will hold the id for the calling process.

Discussion: *Note that process identifiers are dependent on the network interface(s). In particular, if a node has multiple interfaces, it may have multiple node identifiers.*

3.8 Process Aggregation

It is useful in the context of a parallel machine to represent all of the processes in a parallel job through an aggregate identifier. The Portals API provides a mechanism for supporting such job identifiers for these systems. However, job identifiers need not be supported by all systems. In order to be fully supported, job identifiers must be included as a trusted part of a message header, as described in Section 2.3.

The job id is an opaque identifier shared between all of the distributed processes of an application running on a parallel machine. All application processes and job-specific support programs, such as the parallel job launcher, share the same job id. This id is assigned by the runtime system upon job launch, and is guaranteed to be unique among application jobs across the entire distributed system. Individual serial process may be assigned a job id that is not shared with any other processes in the system or the constant `PTL_JID_NONE` can be returned.

Implementations that do not support job ids should return the value `PTL_JID_NONE`.

3.8.1 The Job Id Type

```
int PtlGetJid( ptl_handle_ni_t  ni_handle,
               plt_jid_t       *jid );
```

Return Codes

PTL_OK Indicates success.

PTL_NI_INVALID Indicates the `ni_handle` is not a valid network interface handle.

PTL_NO_INIT Indicates that the Portals API has not been successfully initialized.

PTL_SEGV Indicates that `jid` is not a legal address.

Arguments

<code>ni_handle</code>	input	A network interface handle.
<code>jid</code>	output	On successful return, this location will hold the job id for the calling process.

3.9 Match List Entries and Match Lists

A match list is a chain of match list entries. Each match list entry includes a memory descriptor and a set of match criteria. The match criteria can be used to reject incoming requests based on process id or the match bits provided in the request. A match list is created using the *PtlMEAttach* or *PtlMEAttachAny* functions, which create a match list consisting of a single match list entry, attach the match list to the specified Portal index, and return a handle for the match list entry. Match entries can be dynamically inserted and removed from a match list using the *PtlMEInsert* and *PtlMEUnlink* functions.

3.9.1 Match Entry Type Definitions

```
typedef enum { PTL_RETAIN, PTL_UNLINK } ptl_unlink_t;
typedef enum { PTL_INS_BEFORE, PTL_INS_AFTER } ptl_ins_pos_t;
```

Values of the type `ptl_ins_pos_t` are used to control where a new item is inserted. The value `PTL_INS_BEFORE` is used to insert the new item before the current item or before the head of the list. The value `PTL_INS_AFTER` is used to insert the new item after the current item or after the last item in the list.

3.9.2 PtlMEAttach

```
int PtlMEAttach( ptl_handle_ni_t  ni_handle,
                 ptl_pt_index_t   pt_index,
                 ptl_process_id_t match_id,
                 ptl_match_bits_t match_bits,
                 ptl_match_bits_t ignore_bits,
                 ptl_unlink_t      unlink,
                 ptl_ins_pos_t     position,
                 ptl_handle_me_t* me_handle );
```

The *PtlMEAttach* function creates a match list consisting of a single entry and attaches this list to the Portal table for `ni_handle`.

Return Codes

PTL_OK Indicates success.

PTL_NI_INVALID Indicates that `ni_handle` is not a valid network interface handle.

PTL_NO_INIT Indicates that the Portals API has not been successfully initialized.

PTL_PT_INDEX_INVALID Indicates that `pt_index` is not a valid Portal table index.

PTL_PROCESS_INVALID Indicates that `matchid` is not a valid process identifier.

PTL_NO_SPACE Indicates that there is insufficient memory to allocate the match list entry.

PTL_ME_LIST_TOO_LONG Indicates that the resulting match list is too long. The maximum length for a match list is defined by the interface.

Arguments

<code>ni_handle</code>	input	A handle for the interface to use.
<code>pt_index</code>	input	The Portal table index where the match list should be attached.
<code>match_id</code>	input	Specifies the match criteria for the process id of the requestor. The constants <code>PTL_PID_ANY</code> and <code>PTL_NID_ANY</code> can be used to wildcard either of the ids in the <code>ptl_process_id_t</code> structure.
<code>match_bits, ignore_bits</code>	input	Specify the match criteria to apply to the match bits in the incoming request. The <code>ignore_bits</code> are used to mask out insignificant bits in the incoming match bits. The resulting bits are then compared to the match list entry's match bits to determine if the incoming request meets the match criteria.
<code>unlink</code>	input	Indicates the match list entry should be unlinked when the memory descriptor associated with this match list entry is unlinked. (Note, the check for unlinking a match entry only occurs when the memory descriptor is unlinked.)
<code>position</code>	input	Indicates whether the new match entry should be prepended or appended to the existing match list. If there is no existing list, this argument is ignored and the new match entry becomes the only entry in the list. Allowed constants: <code>PTL_INS_BEFORE</code> , <code>PTL_INS_AFTER</code> .
<code>me_handle</code>	output	On successful return, this location will hold a handle for the newly created match list entry.

3.9.3 PtlMEAttachAny

```
int PtlMEAttachAny( ptl_handle_ni_t  ni_handle,
                    ptl_pt_index_t   *pt_index,
                    ptl_process_id_t match_id,
                    ptl_match_bits_t match_bits,
                    ptl_match_bits_t ignore_bits,
                    ptl_unlink_t      unlink,
                    ptl_handle_me_t* me_handle );
```

The *PtlMEAttachAny* function creates a match list consisting of a single entry and attaches this list to an unused Portal table entry for interface.

Return Codes

PTL_OK Indicates success.

PTL_NI_INVALID Indicates that `interface` is not a valid network interface handle.

PTL_NO_INIT Indicates that the Portals API has not been successfully initialized.

PTL_PROCESS_INVALID Indicates that `matchid` is not a valid process identifier.

PTL_NO_SPACE Indicates that there is insufficient memory to allocate the match list entry.

PTL_PT_FULL Indicates that there are no free entries in the Portal table.

Arguments

<code>ni_handle</code>	input	A handle for the interface to use.
<code>pt_index</code>	output	On succesful return, this location will hold the Portal index where the match list has been attached.
<code>match_id, match_bits, ignore_bits, unlink</code>	input	See the discussion for <i>PtlMEAttach</i> .
<code>me_handle</code>	output	On successful return, this location will hold a handle for the newly created match list entry.

3.9.4 PtlMEInsert

```
int PtlMEInsert( ptl_handle_me_t base,
                 ptl_process_id_t match_id,
                 ptl_match_bits_t match_bits,
                 ptl_match_bits_t ignore_bits,
                 ptl_unlink_t unlink,
                 ptl_ins_pos_t position,
                 ptl_handle_me_t* me_handle );
```

The *PtlMEInsert* function creates a new match list entry and inserts this entry into the match list containing `base`.

Return Codes

PTL_OK Indicates success.

PTL_NO_INIT Indicates that the Portals API has not been successfully initialized.

PTL_PROCESS_INVALID Indicates that `matchid` is not a valid process identifier.

PTL_ME_INVALID Indicates that `base` is not a valid match entry handle.

PTL_ME_LIST_TOO_LONG Indicates that the resulting match list is too long. The maximum length for a match list is defined by the interface.

PTL_NO_SPACE Indicates that there is insufficient memory to allocate the match entry.

Arguments

<code>base</code>	input	A handle for a match entry. The new match entry will be inserted immediately before or immediately after this match entry.
<code>match_id, match_bits, ignore_bits, unlink</code>	input	See the discussion for <i>PtlMEAttach</i>
<code>position</code>	input	Indicates whether the new match entry should be inserted before or after the <code>base</code> entry. Allowed constants: <code>PTL_INS_BEFORE</code> , <code>PTL_INS_AFTER</code> .
<code>me_handle</code>	input	See the discussion for <i>PtlMEAttach</i> .

3.9.5 PtlMEUnlink

```
int PtlMEUnlink( ptl_handle_me_t me_handle );
```

The *PtlMEUnlink* function can be used to unlink a match entry from a match list. This operation also releases any resources associated with the match entry (possibly including the associated memory descriptor). It is an error to use the match entry handle after calling *PtlMEUnlink*.

Return Codes

PTL_OK Indicates success.

PTL_NO_INIT Indicates that the Portals API has not been successfully initialized.

PTL_ME_INVALID Indicates that *me_handle* is not a valid match entry handle.

PTL_ME_IN_USE Indicates that the ME has pending operations and cannot be unlinked.

Arguments

me_handle **input** A handle for the match entry to be unlinked.

3.10 Memory Descriptors

A memory descriptor contains information about a region of a process' memory and optionally an event queue where information about the operations performed on the memory descriptor are recorded. The Portals API provides two operations to create memory descriptors: *PtlMDAttach*, and *PtlMDBind*; an operation to update a memory descriptor, *PtlMDUpdate*; and an operation to unlink and release the resources associated with a memory descriptor, *PtlMDUnlink*.

3.10.1 The Memory Descriptor Type

```
typedef struct {
    void*          start;
    ptl_size_t     length;
    int            threshold;
    unsigned int   max_size;
    unsigned int   options;
    void*          user_ptr;
    ptl_handle_eq_t eq_handle;
} ptl_md_t;
```

The *ptl_md_t* type defines the visible parts of a memory descriptor. Values of this type are used to initialize and update the memory descriptors.

Members

start, length Specify the memory region associated with the memory descriptor. The *start* member specifies the starting address for the memory region and the *length* member specifies the length of the region. The *start* member can be NULL provided that the *length* member is zero. (Zero length buffers are useful to record events.) There are no alignment restrictions on the starting address or the length of the region; although, unaligned messages may be slower (i.e., lower bandwidth and/or longer latency) on some implementations.

threshold Specifies the maximum number of operations that can be performed on the memory descriptor. An operation is any action that could possibly generate an event pair (see Section 3.11.1 for the different types of events). In the usual case, the threshold value is decremented for each operation on the memory descriptor. When the threshold value is zero, the memory descriptor is *inactive*, and does not respond to operations. A memory descriptor can have an initial threshold value of zero to allow for manipulation of an inactive memory descriptor by the local process. A threshold value of `PTL_MD_THRESH_INF` indicates that there is no bound on the number of operations that may be applied to a memory descriptor. Note that local operations (e.g., *PtlMDUpdate*) are not applied to the threshold count.

max_size Specifies the largest incoming request that the memory descriptor will respond to. When the unused portion of a memory descriptor (length - local offset) falls below this value, the memory descriptor becomes *inactive* and does not respond to further operations. This value is only used if the `PTL_MD_MAX_SIZE` option is specified.

options Specifies the behavior of the memory descriptor. The following options can be selected: enable put operations (yes or no), enable get operations (yes or no), offset management (local or remote), message truncation (yes or no), acknowledgement (yes or no), use scatter/gather vectors, disable start events, and disable end events. Values for this argument can be constructed using a bitwise or of the following values:

PTL_MD_OP_PUT Specifies that the memory descriptor will respond to *put* operations. By default, memory descriptors reject *put* operations.

PTL_MD_OP_GET Specifies that the memory descriptor will respond to *get* operations. By default, memory descriptors reject *get* operations.

PTL_MD_MANAGE_REMOTE Specifies that the offset used in accessing the memory region is provided by the incoming request. By default, the offset is maintained locally. When the offset is maintained locally, the offset is incremented by the length of the request so that the next operation (put and/or get) will access the next part of the memory region.

PTL_MD_TRUNCATE Specifies that the length provided in the incoming request can be reduced to match the memory available in the region. (The memory available in a memory region is determined by subtracting the offset from the length of the memory region.) By default, if the length in the incoming operation is greater than the amount of memory available, the operation is rejected.

PTL_MD_ACK_DISABLE Specifies that an acknowledgement should *not* be sent for incoming *put* operations, even if requested. By default, acknowledgements are sent for *put* operations that request an acknowledgement. Acknowledgements are never sent for *get* operations. The data sent in the reply serves as an implicit acknowledgement.

PTL_MD_IOVEC Specifies that the *start* argument is a pointer to an array of type `ptl_md_iovec_t` (see Section 3.10.2) and the *length* argument is the length of the array. This allows for a gather/scatter capability for memory descriptors. A scatter/gather memory descriptor behaves exactly as a memory descriptor that describes a single virtually contiguous region of memory. The local offset, truncation semantics, etc., are identical.

PTL_MD_MAX_SIZE Specifies that the `max_size` field in the memory descriptor is to be used.

PTL_MD_EVENT_START_DISABLE Specifies that this memory descriptor should not generate `PTL_EVENT_*_START` events.

PTL_MD_EVENT_END_DISABLE Specifies that this memory descriptor should not generate `PTL_EVENT_*_END` events.

Note: It is not considered an error to have a memory descriptor that does not respond to either *put* or *get* operations: Every memory descriptor responds to *reply* operations. Nor is it considered an error to have a memory descriptor that responds to both *put* and *get* operations. In fact, a memory descriptor used in a *getput* operation must be configured to respond to both *put* and *get* operations.

user_ptr A user-specified value that is associated with the memory descriptor. The value does not need to be a pointer, but must fit in the space used by a pointer. This value (along with other values) is recorded in events associated with operations on this memory descriptor.¹

eq_handle A handle for the event queue used to log the operations performed on the memory region. If this argument is `PTL_EQ_NONE`, operations performed on this memory descriptor are not logged.

3.10.2 The Memory Descriptor IO Vector Type

```
typedef struct {
    void*      iov_base;
    ptl_size_t iov_len;
} ptl_md_iovec_t;
```

The `ptl_md_iovec_t` type is used to describe gather/scatter buffers of a memory descriptor in conjunction with the `PTL_MD_IOVEC` option. The `ptl_md_iovec_t` is intended to be a type definition of the `struct iovec` type on systems that already support this type.

3.10.3 PtlMDAttach

```
int PtlMDAttach( ptl_handle_me_t me_handle,
                 ptl_md_t      md,
                 ptl_unlink_t   unlink_op,
                 ptl_handle_md_t* md_handle );
```

Values of the type `ptl_unlink_t` are used to control whether an item is unlinked from a list. The value `PTL_UNLINK` enables unlinking. The value `PTL_RETAIN` disables unlinking.

The *PtlMDAttach* operation is used to create a memory descriptor and attach it to a match list entry. An error code is returned if this match list entry already has an associated memory descriptor.

Return Codes

PTL_OK Indicates success.

PTL_NO_INIT Indicates that the Portals API has not been successfully initialized.

PTL_ME_IN_USE Indicates that `me_handle` already has a memory descriptor attached.

PTL_ME_INVALID Indicates that `me_handle` is not a valid match entry handle.

PTL_MD_ILLEGAL Indicates that `md` is not a legal memory descriptor. This may happen because the memory region defined in `md` is invalid or because the network interface associated with the `eq_handle` in `md` is not the same as the network interface associated with `me_handle`.

PTL_EQ_INVALID Indicates that the event queue associated with `md` is not valid.

PTL_NO_SPACE Indicates that there is insufficient memory to allocate the memory descriptor.

PTL_SEGV Indicates that `md_handle` is not a legal address.

¹Tying the memory descriptor to a user-defined value can be useful when multiple memory descriptor share the same event queue or when the memory descriptor needs to be associated with a data structure maintained by the process outside of the Portals library. For example, an MPI implementation can set the `user_ptr` argument to the value of an MPI Request. This direct association allows for processing of memory descriptor's by the MPI implementation without a table lookup or a search for the appropriate MPI Request.

Arguments

<code>me_handle</code>	input	A handle for the match entry that the memory descriptor will be associated with.
<code>md</code>	input	Provides initial values for the user-visible parts of a memory descriptor. Other than its use for initialization, there is no linkage between this structure and the memory descriptor maintained by the API.
<code>unlink_op</code>	input	A flag to indicate whether the memory descriptor is unlinked when it becomes inactive, either because the operation threshold drops to zero or because the <code>max_size</code> threshold value has been exceeded. (Note, the check for unlinking a memory descriptor only occurs after a the completion of a successful operation. If the threshold is set to zero during initialization or using <i>PtlMDUpdate</i> , the memory descriptor is not unlinked.)
<code>md_handle</code>	output	On successful return, this location will hold a handle for the newly created memory descriptor. The <code>handle</code> argument can be NULL, in which case the handle will not be returned.

3.10.4 PtlMDBind

```
int PtlMDBind( ptl_handle_ni_t  ni_handle,
               ptl_md_t         md,
               ptl_unlink_t     unlink_op,
               ptl_handle_md_t* md_handle );
```

The *PtlMDBind* operation is used to create a “free floating” memory descriptor, i.e., a memory descriptor that is not associated with a match list entry.

Return Codes

PTL_OK Indicates success.

PTL_NO_INIT Indicates that the Portals API has not been successfully initialized.

PTL_NI_INVALID Indicates that `ni_handle` is not a valid network interface handle.

PTL_MD_ILLEGAL Indicates that `md` is not a legal memory descriptor. This may happen because the memory region defined in `md` is invalid or because the network interface associated with the `eq_handle` in `md` is not the same as the network interface, `ni_handle`.

PTL_EQ_INVALID Indicates that the event queue associated with `md` is not valid.

PTL_NO_SPACE Indicates that there is insufficient memory to allocate the memory descriptor.

PTL_SEGV Indicates that `md_handle` is not a legal address.

Arguments

<code>ni_handle</code>	input	A handle for the network interface with which the memory descriptor will be associated.
<code>md, unlink_op</code>	input	See the discussion for <i>PtlMDAttach</i> .
<code>md_handle</code>	output	On successful return, this location will hold a handle for the newly created memory descriptor. The <code>handle</code> argument must be a valid address and cannot be NULL.

3.10.5 PtlMDUnlink

```
int PtlMDUnlink( ptl_handle_md_t md_handle );
```

The *PtlMDUnlink* function unlinks the memory descriptor from any match list entry it may be linked to and releases the resources associated with a memory descriptor. (This function does not free the memory region associated with the memory descriptor.) This function also releases the resources associated with a floating memory descriptor. Only memory descriptors with no pending operations may be unlinked. Explicitly unlinking a memory descriptor via this function call has the same behavior as a memory descriptor that has been automatically unlinked, except that a `PTL_EVENT_UNLINK` event is not generated.

Return Codes

PTL_OK Indicates success.

PTL_NO_INIT Indicates that the Portals API has not been successfully initialized.

PTL_MD_INVALID Indicates that `md_handle` is not a valid memory descriptor handle.

PTL_MD_IN_USE Indicates that `md_handle` has pending operations and cannot be unlinked.

Arguments

`md_handle` **input** A handle for the memory descriptor to be released.

3.10.6 PtlMDUpdate

```
int PtlMDUpdate( ptl_handle_md_t md_handle,
                 ptl_md_t*      old_md,
                 ptl_md_t*      new_md,
                 ptl_handle_eq_t eq_handle );
```

The *PtlMDUpdate* function provides a conditional, atomic update operation for memory descriptors. The memory descriptor identified by `md_handle` is only updated if the event queue identified by `eq_handle` is empty. The intent is to only enable updates to the memory descriptor when no new messages have arrived since the last time the queue was checked.

If `new` is not NULL the memory descriptor identified by `md_handle` will be updated to reflect the values in the structure pointed to by `new` if `eq_handle` has the value `PTL_EQ_NONE` or if the event queue identified by `eq_handle` is empty. If `old` is not NULL, the current value of the memory descriptor identified by `md_handle` is recorded in the location identified by `old`.

Return Codes

PTL_OK Indicates success.

PTL_NO_INIT Indicates that the Portals API has not been successfully initialized.

PTL_MD_NO_UPDATE Indicates that the update was not performed because `eq_handle` was not empty.

PTL_MD_INVALID Indicates that `md_handle` is not a valid memory descriptor handle.

PTL_MD_ILLEGAL Indicates that the value pointed to by `new` is not a legal memory descriptor (e.g., the memory region specified by the memory descriptor may be invalid).

PTL_EQ_INVALID Indicates that `eq_handle` is not a valid event queue handle.

PTL_SEGV Indicates that `new` or `old` is not a legal address.

Arguments

<code>md_handle</code>	input	A handle for the memory descriptor to update.
<code>old_md</code>	output	If <code>old_md</code> is not the value <code>NULL</code> , the current value of the memory descriptor will be stored in the location identified by <code>old_md</code> .
<code>new_md</code>	input	If <code>new_md</code> is not the value <code>NULL</code> , this argument provides the new values for the memory descriptor, if the update is performed.
<code>eq_handle</code>	input	A handle for an event queue used to predicate the update. If <code>eq_handle</code> is equal to <code>PTL_EQ_NONE</code> , the update is performed unconditionally. Otherwise, the update is performed if and only if <code>eq_handle</code> is empty. If the update is not performed, the function returns the value <code>PTL_NO_UPDATE</code> . (Note, the <code>eq_handle</code> argument does not need to be the same as the event queue associated with the memory descriptor.)

The conditional update can be used to ensure that the memory descriptor has not changed between the time it was examined and the time it is updated. In particular, it is needed to support an MPI implementation where the activity of searching an unexpected message queue and posting a receive must be atomic.

3.10.7 Thresholds and Unlinking

The value of the threshold is checked before each operation. If the threshold is nonzero, it is decremented after the operation is initiated. A threshold that has been decremented to zero may still have operations that are pending. If the MD is configured to automatically unlink, the unlink event will not be generated until all pending operations have been completed. Binding a new MD to an ME is only permitted after the MD has been explicitly unlinked or after an unlink event has been posted.

3.11 Events and Event Queues

Event queues are used to log operations performed on memory descriptors. They can also be used to hold acknowledgements for completed *put* operations and to note when the data specified in a *put* operation has been sent (i.e., when it is safe to reuse the buffer that holds this data). Multiple memory descriptors can share a single event queue. An event queue may have an optional event handler associated with it. If an event handler exists, it will be run for each event that is deposited into the event queue.

In addition to the `ptl_handle_eq_t` type, the Portals API defines two types associated with events: The `ptl_event_kind_t` type defines the kinds of events that can be stored in an event queue. The `ptl_event_t` type defines a structure that holds the information associated with an event.

The Portals API also provides five functions for dealing with event queues: The *PtlEQAlloc* function is used to allocate the API resources needed for an event queue, the *PtlEQFree* function is used to release these resources, the *PtlEQGet* function can be used to get the next event from an event queue, the *PtlEQWait* function can be used to block a process (or thread) until an event queue has at least one event, and the *PtlEQPoll* function can be used to test or wait on multiple event queues.

3.11.1 Kinds of Events

```
typedef enum {
    PTL_EVENT_GET_START, PTL_EVENT_GET_END,
    PTL_EVENT_PUT_START, PTL_EVENT_PUT_END,
    PTL_EVENT_GETPUT_START, PTL_EVENT_GETPUT_END,
    PTL_EVENT_REPLY_START, PTL_EVENT_REPLY_END,
    PTL_EVENT_SEND_START, PTL_EVENT_SEND_END,
    PTL_EVENT_ACK
} ptl_event_kind_t;
```

The Portals API defines fourteen types of events that can be logged in an event queue:

PTL_EVENT_PUT_START A remote *put* operation has been started on the memory descriptor. The memory region associated with this descriptor should be considered volatile until the corresponding END or event is logged.

PTL_EVENT_PUT_END A previously initiated *put* operation completed successfully. The underlying layers will not alter the memory (on behalf of this operation) once this event has been logged.

PTL_EVENT_GETPUT_START A remote *getput* operation has been started on the memory descriptor. The memory region associated with this descriptor should not be altered until the corresponding END event is logged.

PTL_EVENT_GETPUT_END A previously initiated *getput* operation completed successfully.

PTL_EVENT_REPLY_START A *reply* operation has been started on the memory descriptor.

PTL_EVENT_REPLY_END A previously initiated *reply* operation has completed successfully. This event is logged after the data (if any) from the reply has been written into the memory descriptor.

PTL_EVENT_SEND_START An outgoing *send* operation has been started. The memory region associated with this descriptor should not be altered until the corresponding END or event is logged.

PTL_EVENT_SEND_END A previously initiated *send* operation has completed successfully. This event is logged after the entire buffer has been sent and it is safe for the caller to reuse the buffer.

PTL_EVENT_ACK An *acknowledgement* was received. This event is logged when the acknowledgement is received

3.11.2 Event Ordering

As implied by the naming convention, start events must be delivered before end events for a given operation. The Portals API also guarantees that when a process initiates two operations on a remote process, the operations will be started on the remote process in the same order that they were initiated on the origin process. As an example, if process A initiates two *put* operations, *x* and *y*, on process B, the Portals API guarantees that process A will receive the PTL_EVENT_SEND_START events for *x* and *y* in the same order that process B receives the PTL_EVENT_PUT_START events for *x* and *y*.

Note that memory descriptors that have chosen to ignore start or end events using the MD_EVENT_START_DISABLE or MD_EVENT_END_DISABLE options are still subject to ordering constraints. Even if the destination memory descriptors for messages *x* and *y* have chosen to disable all events, messages *x* and *y* must still traverse the Portals data structures (eg. the match list) in the order in which they were initiated.

3.11.3 Failure Notification

Operations may fail to complete successfully; however, unless the node itself fails, every operation that is started will eventually complete. While an operation is in progress, the memory associated with the operation should not be viewed (in the case of a put or a reply) or altered (in the case of a send or get). Operation completion, whether successful or unsuccessful, is final. That is, when an operation completes, the memory associated with the operation will no longer be read or altered by the operation. A network interface can use the integral type `ptl_ni_fail_t` to define specific information regarding the failure of the operation and record this information in the `ni_fail_type` field of an event. The constant `PTL_NI_OK` should be used in successful start and end events to indicate that there has been no failure.

3.11.4 The Event Type

```
typedef struct {
    ptl_event_kind_t    type;
    ptl_process_id_t    initiator;
    ptl_uid_t           uid;
    ptl_jid_t           jid;
```

```

    ptl_pt_index_t      pt_index;
    ptl_match_bits_t    match_bits;
    ptl_size_t          rlength;
    ptl_size_t          mlength;
    ptl_size_t          offset;
    ptl_handle_md_t     md_handle;
    ptl_md_t            md;
    ptl_hdr_data_t      hdr_data;
    ptl_seq_t           link;
    ptl_ni_fail_t       ni_fail_type;
    volatile ptl_seq_t  sequence;
} ptl_event_t;

```

An event structure includes the following members:

type Indicates the type of the event.

initiator The id of the initiator.

pt_index The Portal table index specified in the request.

match_bits A copy of the match bits specified in the request. See section 3.9 for more information on match bits.

rlength The length (in bytes) specified in the request.

mlength The length (in bytes) of the data that was manipulated by the operation. For truncated operations, the manipulated length will be the number of bytes specified by the memory descriptor (possibly with an offset) operation. For all other operations, the manipulated length will be the length of the requested operation.

offset Is the displacement (in bytes) into the memory region that the operation used. The offset can be determined by the operation (see Section 3.13) for a remote managed memory descriptor, or by the local memory descriptor (see Section 3.10). The offset and the length of the memory descriptor can be used to determine if the `max_size` has been exceeded.

md_handle Is the handle to the memory descriptor associated with the event.

md Is the state of the memory descriptor immediately after the event has been processed. In particular, the `threshold` field in `md` will reflect the state of the threshold *after* the operation occurred.

hdr_data 64 bits of out-of-band user data (see Section 3.13.2).

link The *link* member is used to link START events with the END event that signifies completion of the operation. The *link* member will be the same for the two events associated with an operation. The link member is also used to link an UNLINK event with the event that caused the memory descriptor to be unlinked.

ni_fail_type Is used to convey the failure of an operation. See Section 3.11.3.

sequence The sequence number for this event. Sequence numbers are unique to each event.

Discussion: *The sequence member is the last member and is volatile to support SMP implementations. When an event structure is filled in, the sequence member should be written after all other members have been updated. Moreover, a memory barrier should be inserted between the updating of other members and the updating of the sequence member.*

3.11.5 The Event Queue Handler Type

```
typedef void (*ptl_eq_handler_t)( ptl_event_t *event );
```

The `ptl_eq_handler_t` type is used to represent event handler functions.

3.11.6 PtlEQAlloc

```
int PtlEQAlloc( ptl_handle_ni_t  ni_handle,
                ptl_size_t      count,
                ptl_eq_handler_t eq_handler,
                ptl_handle_eq_t* eq_handle );
```

The *PtlEQAlloc* function is used to build an event queue.

Return Codes

PTL_OK Indicates success.

PTL_NO_INIT Indicates that the Portals API has not been successfully initialized.

PTL_NI_INVALID Indicates that *ni_handle* is not a valid network interface handle.

PTL_NO_SPACE Indicates that there is insufficient memory to allocate the event queue.

PTL_SEGV Indicates that *eq_handle* is not a legal address.

Arguments

<i>ni_handle</i>	input	A handle for the interface with which the event queue will be associated.
<i>count</i>	input	A hint as to the number of events to be stored in the event queue. An implementation may provide space for more than the requested number of event queue slots.
<i>eq_handler</i>	input	A handler function that runs when an event is deposited into the event queue. The constant value PTL_EQ_HANDLER_NONE can be used to indicate that no event handler is desired.
<i>eq_handle</i>	output	On successful return, this location will hold a handle for the newly created event queue.

3.11.7 Event Queue Handler Semantics

The event queue handler, if specified, runs for each event that is deposited into the event queue. The handler is supplied with a pointer to the event that triggered the handler invocation. The handler is invoked at some time between when the event is deposited into the event queue by the underlying communication system, and the return of a successful *PtlEQGet*, *PtlEQWait*, or *PtlEQPoll* operation. This implies that if *handler* is not **PTL_EQ_HANDLER_NONE**, *PtlEQGet*, *PtlEQWait*, or *PtlEQPoll* must be called for each event in the queue.

Event handlers may have implementation specific restrictions. In general, handlers must:

- not block,
- not make system calls,
- be reentrant
- not call *PtlEQWait*, *PtlEQGet*, or *PtlEQPoll*
- not perform I/O operations
- be allowed to call the data movement functions (*PtlPut*, *PtlPutRegion*, *PtlGet*, *PtlGetRegion*, *PtlGetPut*).

Discussion: An event handler can be called by the implementation when delivering an event, or by the Portals library when an event is received. In the former case, the implementation must ensure that the address mappings are properly set up for the handler to run. The handler belongs to the address space of the execution thread that called *PtlEQAlloc*. When run, the handler should not receive any privileges it would not have had, if run by the caller of *PtlEQAlloc*.

If handlers are implemented inside the Portals library they must be called before `PtlEQGet`, `PtlEQWait()`, or `PtlEQPoll` returns with a status of `PTL_OK` or `PTL_EQ_DROPPED`. Independent of the implementation, after a successful handler run, the corresponding event in the event queue is removed.

Behavior is undefined if the handler argument to `PtlEQAlloc` is not `PTL_EQ_HANDLER_NONE`, but `PtlEQGet`, `PtlEQWait`, or `PtlEQPoll` are not called for every event in the event queue. Behavior is also undefined if a handler does not follow the implementation specific restrictions, for example if a handler blocks.

3.11.8 PtlEQFree

```
int PtlEQFree( ptl_handle_eq_t eq_handle );
```

The *PtlEQFree* function releases the resources associated with an event queue. It is up to the user to insure that no memory descriptors are associated with the event queue once it is freed.

Return Codes

PTL_OK Indicates success.

PTL_NO_INIT Indicates that the Portals API has not been successfully initialized.

PTL_EQ_INVALID Indicates that `eq_handle` is not a valid event queue handle.

Arguments

`eq_handle` **input** A handle for the event queue to be released.

3.11.9 PtlEQGet

```
int PtlEQGet( ptl_handle_eq_t eq_handle,
              ptl_event_t*   event );
```

The *PtlEQGet* function is a nonblocking function that can be used to get the next event in an event queue. If an event handler is associated with the event queue, then the handler will run before this function returns successfully. The event is removed from the queue.

Return Codes

PTL_OK Indicates success.

PTL_EQ_DROPPED Indicates success (i.e., an event is returned) and that at least one event between this event and the last event obtained (using *PtlEQGet*, *PtlEQWait*, or *PtlEQPoll*) from this event queue has been dropped due to limited space in the event queue.

PTL_NO_INIT Indicates that the Portals API has not been successfully initialized.

PTL_EQ_EMPTY Indicates that `eq_handle` is empty or another thread is waiting on *PtlEQWait*.

PTL_EQ_INVALID Indicates that `eq_handle` is not a valid event queue handle.

PTL_SEGV Indicates that `event` is not a legal address.

Arguments

`eq_handle` **input** A handle for the event queue.
`event` **output** On successful return, this location will hold the values associated with the next event in the event queue.

3.11.10 PtlEQWait

```
int PtlEQWait( ptl_handle_eq_t eq_handle,
               ptl_event_t*   event );
```

The *PtlEQWait* function can be used to block the calling process (thread) until there is an event in an event queue. If an event handler is associated with the event queue, then the handler will run before this function returns successfully. This function returns the next event in the event queue and removes this event from the queue. In the event that multiple threads are waiting on the same event queue, *PtlEQWait* is guaranteed to wake exactly one thread, but the order in which they are awakened is not specified.

Return Codes

PTL_OK Indicates success.

PTL_EQ_DROPPED Indicates success (i.e., an event is returned) and that at least one event between this event and the last event obtained (using *PtlEQGet*, *PtlEQWait*, or *PtlEQPoll*) from this event queue has been dropped due to limited space in the event queue.

PTL_NO_INIT Indicates that the Portals API has not been successfully initialized.

PTL_EQ_INVALID Indicates that *eq_handle* is not a valid event queue handle.

PTL_SEGV Indicates that *event* is not a legal address.

Arguments

<i>eq_handle</i>	input	A handle for the event queue to wait on. The calling process (thread) will be blocked until the event queue is not empty.
<i>event</i>	output	On successful return, this location will hold the values associated with the next event in the event queue.

3.11.11 PtlEQPoll

```
int PtlEQPoll( ptl_handle_eq_t* eq_handles,
               int               size,
               int               timeout,
               ptl_event_t*      event,
               int*              which );
```

The *PtlEQPoll* function can be used by the calling process to look for an event from a set of event queues. Should an event arrive on any of the queues contained in the array of event queue handles, the event will be returned in *event* and *which* will contain the index of the event queue from which the event was taken. If an event handler is associated with the event queue, then the handler will run before this function returns successfully. *PtlEQPoll* provides a timeout to allow applications to poll, block for a fixed period, or block indefinitely. *PtlEQPoll* is sufficiently general to implement both *PtlEQGet* and *PtlEQWait*, but these functions have been retained in the API for backward compatibility.

Return Codes

PTL_OK Indicates success.

PTL_EQ_DROPPED Indicates success (i.e., an event is returned) and that at least one event between this event and the last event obtained from the event queue indicated by *which* has been dropped due to limited space in the event queue.

PTL_NO_INIT Indicates that the Portals API has not been successfully initialized.

PTL_EQ_INVALID Indicates that one or more of the event queue handles is not a valid.

PTL_SEGV Indicates that `event` or `which` is not a legal address.

PTL_EQ_EMPTY Indicates that the timeout has been reached and all of the event queues are empty.

Arguments

<code>eq_handles</code>	input	An array of event queue handles.
<code>size</code>	input	Length of the array.
<code>timeout</code>	input	Time in milliseconds to wait for an event to occur on one of the event queue handles. The constant <code>PTL_TIME_FOREVER</code> can be used to indicate an infinite timeout.
<code>event</code>	output	On successful return (<code>PTL_OK</code> or <code>PTL_EQ_DROPPED</code>), this location will hold the values associated with the next event in the event queue.
<code>which</code>	output	On successful return, this location will contain the index of the event queue from which the event was taken.

3.11.12 Event Semantics

The split event sequence is needed to support unreliable networks and/or networks that packetize. The start/end sequence is needed to support networks that packetize where the completion of transfers may not be ordered with initiation of transfers. An implementation is free to implement these event sequences in any way that meets the ordering semantics. For example, an implementation for a network that is reliable and that preserves message ordering (or does not packetize) may generate a start/end event pair at the completion of the transfer. In fact, since the information in the start/end events is identical, except for the link field, a correct implementation may generate a single event that the EQ test/wait library function turns into an event pair.

All of the members of the `ptl_event_t` structure returned from *PtlEQGet* and *PtlEQWait* must be filled in with valid information. An implementation may not leave any field in an event unset.

3.12 The Access Control Table

Processes can use the access control table to control which processes are allowed to perform operations on Portal table entries. Each communication interface has a Portal table and an access control table. The access control table for the default interface contains an entry at index zero that allows all processes with the same user id to communicate. Entries in the access control table can be manipulated using the *PtlACEntry* function.

3.12.1 PtlACEntry

```
int PtlACEntry( ptl_handle_ni_t  ni_handle,
                ptl_ac_index_t   ac_index,
                ptl_process_id_t match_id,
                ptl_uid_t        user_id,
                ptl_jid_t        job_id,
                ptl_pt_index_t    pt_index );
```

The *PtlACEntry* function can be used to update an entry in the access control table for an interface. For those implementations that do not support job identifiers, the `job_id` argument is ignored.

Return Codes

PTL_OK Indicates success.

PTL_NO_INIT Indicates that the Portals API has not been successfully initialized.

PTL_NI_INVALID Indicates that `ni_handle` is not a valid network interface handle.

PTL_AC_INDEX_INVALID Indicates that `ac_index` is not a valid access control table index.

PTL_PROCESS_INVALID Indicates that `match_id` is not a valid process identifier.

PTL_PT_INDEX_INVALID Indicates that `pt_index` is not a valid Portal table index.

Arguments

<code>ni_handle</code>	input	Identifies the interface to use.
<code>ac_index</code>	input	The index of the entry in the access control table to update.
<code>match_id</code>	input	Identifies the process(es) that are allowed to perform operations. The constants <code>PTL_PID_ANY</code> and <code>PTL_NID_ANY</code> can be used to wildcard either of the ids in the <code>ptl_process_id_t</code> structure.
<code>user_id</code>	input	Identifies the user that is allowed to perform operations. The value <code>PTL_UID_ANY</code> can be used to wildcard the user.
<code>job_id</code>	input	Identifies the collection of processes allowed to perform an operation. The value <code>PTL_JID_ANY</code> can be used to wildcard the job id.
<code>pt_index</code>	input	Identifies the Portal index(es) that can be used. The value <code>PTL_PT_INDEX_ANY</code> can be used to wildcard the Portal index.

3.13 Data Movement Operations

The Portals API provides five data movement operations: *PtlPut*, *PtlPutRegion*, *PtlGet*, *PtlGetRegion*, and *PtlGetPut*.

3.13.1 Portal Acknowledgment Type Definition

```
typedef enum { PTL_ACK_REQ, PTL_NO_ACK_REQ } ptl_ack_req_t;
```

Values of the type `ptl_ack_req_t` are used to control whether an acknowledgement should be sent when the operation completes (i.e., when the data has been written to a memory descriptor of the target process). The value `PTL_ACK_REQ` requests an acknowledgement, the value `PTL_NO_ACK_REQ` requests that no acknowledgement should be generated.

3.13.2 PtlPut

```
int PtlPut( ptl_handle_md_t  md_handle,
            ptl_ack_req_t    ack_req,
            ptl_process_id_t target_id,
            ptl_pt_index_t   pt_index,
            ptl_ac_index_t   ac_index,
            ptl_match_bits_t match_bits,
            ptl_size_t        remote_offset,
            ptl_hdr_data_t    hdr_data );
```

The *PtlPut* function initiates an asynchronous put operation. There are several events associated with a put operation: initiation of the send on the local node (`PTL_EVENT_SEND_START`), completion of the send on the local node (`PTL_EVENT_SEND_END`), and, when the send completes successfully, the receipt of an acknowledgement (`PTL_EVENT_ACK`) indicating that the operation was accepted by the target. These events will be logged in the event queue associated with the memory descriptor (`md_handle`) used in the put operation. Using a memory descriptor that does not have an associated event queue results in these events being discarded. In this case, the caller must have another mechanism (e.g., a higher level protocol) for determining when it is safe to modify the memory region associated with the memory descriptor.

Return Codes

PTL_OK Indicates success.

PTL_NO_INIT Indicates that the Portals API has not been successfully initialized.

PTL_MD_INVALID Indicates that `md_handle` is not a valid memory descriptor.

PTL_PROCESS_INVALID Indicates that `target_id` is not a valid process id.

Arguments

<code>md_handle</code>	input	A handle for the memory descriptor that describes the memory to be sent. If the memory descriptor has an event queue associated with it, it will be used to record events when the message has been sent (<code>PTL_EVENT_SEND_START</code> , <code>PTL_EVENT_SEND_END</code>).
<code>ack_req</code>	input	Controls whether an acknowledgement event is requested. Acknowledgements are only sent when they are requested by the initiating process and the memory descriptor has an event queue and the target memory descriptor enables them. Allowed constants: <code>PTL_ACK_REQ</code> , <code>PTL_NO_ACK_REQ</code> .
<code>target_id</code>	input	A process id for the target process.
<code>pt_index</code>	input	The index in the remote Portal table.
<code>ac_index</code>	input	The index into the access control table of the target process.
<code>match_bits</code>	input	The match bits to use for message selection at the target process.
<code>remote_offset</code>	input	The offset into the target memory descriptor (only used when the target memory descriptor has the <code>PTL_MD_MANAGE_REMOTE</code> option set).
<code>hdr_data</code>	input	64 bits of user data that can be included in message header. This data is written to an event queue entry at the target if an event queue is present on the matching memory descriptor.

3.13.3 PtlPutRegion

```
int PtlPutRegion( ptl_handle_md_t  md_handle,
                  ptl_size_t       local_offset,
                  ptl_size_t       length;
                  ptl_ack_req_t    ack_req,
                  ptl_process_id_t target_id,
                  ptl_pt_index_t   pt_index,
                  ptl_ac_index_t   ac_index,
                  ptl_match_bits_t match_bits,
                  ptl_size_t       remote_offset,
                  ptl_hdr_data_t   hdr_data );
```

The *PtlPutRegion* function is identical to the *PltPut* function except that it allows a region of memory within the memory descriptor to be sent rather than the entire memory descriptor. The local offset is used to determine the starting address of the memory region and the length specifies the length of the region in bytes. It is an error for the local offset and length parameters to specify memory outside the memory described by the memory descriptor.

Return Codes

PTL_OK Indicates success.

PTL_NO_INIT Indicates that the Portals API has not been successfully initialized.

PTL_MD_INVALID Indicates that `md_handle` is not a valid memory descriptor.

PTL_MD_ILLEGAL Indicates that `local_offset` and `length` specify a region outside the bounds of the memory descriptor.

PTL_PROCESS_INVALID Indicates that `target_id` is not a valid process id.

Arguments

<code>md_handle</code>	input	A handle for the memory descriptor that describes the memory to be sent.
<code>local_offset</code>	input	Offset from the start of the memory descriptor.
<code>length</code>	input	Length of the memory region to be sent.
<code>ack_req, target_id, pt_index, ac_index</code>	input	See the discussion for <i>PtlPut</i> .
<code>match_bits, remote_offset, hdr_data</code>	input	See the discussion for <i>PtlPut</i> .

3.13.4 PtlGet

```
int PtlGet( ptl_handle_md_t  md_handle,
            ptl_process_id_t target_id,
            ptl_pt_index_t   pt_index,
            ptl_ac_index_t   ac_index,
            ptl_match_bits_t match_bits,
            ptl_size_t        remote_offset );
```

The *PtlGet* function initiates a remote read operation. There are two event pairs associated with a get operation, when the data is sent from the remote node, a `PTL_EVENT_GET_{START,END}` event pair is registered on the remote node; and when the data is returned from the remote node a `PTL_EVENT_REPLY_{START,END}` event pair is registered on the local node.

Return Codes

PTL_OK Indicates success.

PTL_NO_INIT Indicates that the Portals API has not been successfully initialized.

PTL_MD_INVALID Indicates that `md_handle` is not a valid memory descriptor.

PTL_PROCESS_INVALID Indicates that `target_id` is not a valid process id.

Arguments

<code>md_handle</code>	input	A handle for the memory descriptor that describes the memory into which the requested data will be received. The memory descriptor can have an event queue associated with it to record events, such as when the message receive has started.
<code>target_id</code>	input	A process id for the target process.
<code>pt_index</code>	input	The index in the remote Portal table.
<code>ac_index</code>	input	The index into the access control table of the target process.
<code>match_bits</code>	input	The match bits to use for message selection at the target process.
<code>remote_offset</code>	input	The offset into the target memory descriptor (only used when the target memory descriptor has the <code>PTL_MD_MANAGE_REMOTE</code> option set).

3.13.5 PtlGetRegion

```
int PtlGetRegion( ptl_handle_md_t  md_handle,
                  ptl_size_t        local_offset,
                  ptl_size_t        length,
                  ptl_process_id_t target_id,
```

```

    ptl_pt_index_t    pt_index,
    ptl_ac_index_t    ac_index,
    ptl_match_bits_t  match_bits,
    ptl_size_t        remote_offset );

```

The *PtlGetRegion* function is identical to the *PtlGet* function except that it allows a region of memory within the memory descriptor to accept a reply rather than the entire memory descriptor. The local offset is used to determine the starting address of the memory region and the length specifies the length of the region in bytes. It is an error for the local offset and length parameters to specify memory outside the memory described by the memory descriptor.

Return Codes

PTL_OK Indicates success.

PTL_NO_INIT Indicates that the Portals API has not been successfully initialized.

PTL_MD_INVALID Indicates that *md_handle* is not a valid memory descriptor.

PTL_MD_ILLEGAL Indicates that *local_offset* and *length* specify a region outside the bounds of the memory descriptor.

PTL_PROCESS_INVALID Indicates that *target_id* is not a valid process id.

Arguments

<i>md_handle</i>	input	A handle for the memory descriptor that describes the memory into which the requested data will be received. The memory descriptor can have an event queue associated with it to record events, such as when the message receive has started.
<i>local_offset</i>	input	Offset from the start of the memory descriptor.
<i>length</i>	input	Length of the memory region for the reply.
<i>target_id, pt_index, ac_index</i>	input	See discussion for <i>PtlGet</i> .
<i>match_bits, remote_offset</i>	input	See discussion for <i>PtlGet</i> .

3.13.6 PtlGetPut

```

int PtlGetPut( ptl_handle_md_t  get_md_handle,
               ptl_handle_md_t  put_md_handle,
               ptl_process_id_t  target_id,
               ptl_pt_index_t    pt_index,
               ptl_ac_index_t    ac_index,
               ptl_match_bits_t  match_bits,
               ptl_size_t        remote_offset,
               ptl_hdr_data_t    hdr_data );

```

The *PtlGet* function performs an atomic swap of data at the destination with the data passed in the *put* memory descriptor. The original contents of the memory region on the target are returned in a reply message and placed into the *get* memory descriptor upon receipt by the initiator. An implementation may restrict the length of the memory descriptors used in *PtlGetPut*, but must support at least 8 bytes. The remote memory descriptor must be configured to respond to both *get* operations and *put* operations.

There are three event pairs associated with a get operation. When data is sent from the local node, a PTL_EVENT_SENT_{START,END} is registered on the local node. When the data is sent from the remote node, a PTL_EVENT_GETPUT_{START,END} event pair is registered on the remote node; and when the data is returned from the remote node a PTL_EVENT_REPLY_{START,END} event pair is registered on the local node.

Return Codes

PTL_OK Indicates success.

PTL_NO_INIT Indicates that the Portals API has not been successfully initialized.

PTL_MD_INVALID Indicates that `put_md_handle` or `get_md_handle` is not a valid memory descriptor.

PTL_PROCESS_INVALID Indicates that `target_id` is not a valid process id.

Arguments

<code>get_md_handle</code>	input	A handle for the memory descriptor that describes the memory into which the requested data will be received. The memory descriptor can have an event queue associated with it to record events, such as when the message receive has started.
<code>put_md_handle</code>	input	A handle for the memory descriptor that describes the memory to be sent. If the memory descriptor has an event queue associated with it, it will be used to record events when the message has been sent.
<code>target_id</code>	input	A process id for the target process.
<code>pt_index</code>	input	The index in the remote Portal table.
<code>ac_index</code>	input	The index into the access control table of the target process.
<code>match_bits</code>	input	The match bits to use for message selection at the target process.
<code>remote_offset</code>	input	The offset into the target memory descriptor (only used when the target memory descriptor has the <code>PTL_MD_MANAGE_REMOTE</code> option set).
<code>hdr_data</code>	input	64 bits of user data that can be included in message header. This data is written to an event queue entry at the target if an event queue is present on the matching memory descriptor.

3.14 PtlHandleIsEqual

```
int PtlHandleIsEqual( ptl_handle_any_t  handle1,
                     ptl_handle_any_t  handle2);
```

The *PtlHandleIsEqual* function compares two handles to determine if they represent the same object.

Return Codes

PTL_OK Indicates that the handles are equivalent.

PTL_FAIL Indicates that the two handles are not equivalent.

Arguments

<code>handle1, handle2</code>	input	A handle for an object. Either of these handles is allowed to be the constant value, <code>PTL_INVALID_HANDLE</code> , which represents the value of an invalid handle.
-------------------------------	--------------	---

3.15 Summary

We conclude this section by summarizing the names introduced by the Portals 3.3 API. We start by summarizing the names of the types introduced by the API. This is followed by a summary of the functions introduced by the API. Which is followed by a summary of the function return codes. Finally, we conclude with a summary of the other constant values introduced by the API.

Table 3.2 presents a summary of the types defined by the Portals API. The first column in this table gives the type name, the second column gives a brief description of the type, the third column identifies the section where the type is defined, and the fourth column lists the functions that have arguments of this type.

Table 3.2: Types Defined by the Portals 3.3 API

Name	Meaning	Sect	Functions
ptl_ac_index_t	indexes for an access control table	3.2.3	PtlACEEntry, PtlPut, PtlPutRegion, PtlGet, PtlGetRegion, PtlGetPut
ptl_ack_req_t	acknowledgement request types	3.13.2	PtlPut, PtlPutRegion
ptl_eq_handler_t	event queue handler function	3.11.5	PtlEQAlloc
ptl_event_kind_t	kinds of events	3.11.1	PtlEQGet, PtlEQWait, PtlEQPoll
ptl_event_t	information about events	3.11.4	PtlEQGet, PtlEQWait, PtlEQPoll
ptl_seq_t	event sequence number	3.11.4	PtlEQGet, PtlEQWait, PtlEQPoll
ptl_handle_any_t	handles for any object	3.2.2	PtlNIHandle, PtlHandleIsEqual
ptl_handle_eq_t	handles for event queues	3.2.2	PtlEQAlloc, PtlEQFree, PtlEQGet, PtlEQWait, PtlEQPoll, PtlMDUpdate
ptl_handle_md_t	handles for memory descriptors	3.2.2	PtlMDAlloc, PtlMDUnlink, PtlMDUpdate, PtlMEAttach, PtlMEAttachAny, PtlMEInsert, PtlPut, PtlPutRegion, PtlGet, PtlGetRegion, PtlGetPut
ptl_handle_me_t	handles for match entries	3.2.2	PtlMEAttach, PtlMEAttachAny, PtlMEInsert, PtlMEUnlink
ptl_handle_ni_t	handles for network interfaces	3.2.2	PtlNIInit, PtlNIFini, PtlNISStatus, PtlNIDist, PtlEQAlloc, PtlACEEntry
ptl_nid_t	node identifiers	3.2.6	PtlGetId, PtlACEEntry
ptl_pid_t	process identifier	3.2.6	PtlGetId, PtlACEEntry
ptl_uid_t	user identifier	3.2.6	PtlGetUid, PtlACEEntry
ptl_jid_t	job identifier	3.8	PtlGetJid, PtlACEEntry
ptl_ins_pos_t	insertion position (before or after)	3.9.2	PtlMEAttach, PtlMEAttachAny, PtlMEInsert
ptl_interface_t	identifiers for network interfaces	3.2.5	PtlNIInit
ptl_match_bits_t	match (and ignore) bits	3.2.4	PtlMEAttach, PtlMEAttachAny, PtlMEInsert, PtlPut, PtlPutRegion, PtlGet, PtlGetRegion, PtlGetPut
ptl_md_t	memory descriptors	3.10.1	PtlMDAttach, PtlMDBind, PtlMDUpdate
ptl_ni_fail_t	network interface-specific failures	3.11	PtlEQGet, PtlEQWait, PtlEQPoll
ptl_process_id_t	process identifiers	3.7.1	PtlGetId, PtlNIDist, PtlMEAttach, PtlMEAttachAny, PtlACEEntry, PtlPut, PtlPutRegion, PtlGet, PtlGetRegion, PtlGetPut
ptl_pt_index_t	indexes for Portal tables	3.2.3	PtlMEAttach, PtlMEAttachAny, PtlPut, PtlPutRegion, PtlGet, PtlGetRegion, PtlGetPut, PtlACEEntry
ptl_size_t	sizes	3.2.1	PtlEQAlloc, PtlPut, PtlPutRegion, PtlGet, PtlGetRegion
ptl_sr_index_t	indexes for status registers	3.2.7	PtlNISStatus
ptl_sr_value_t	values in status registers	3.2.7	PtlNISStatus
ptl_unlink_t	unlink options	3.9.2	PtlMEAttach, PtlMEAttachAny, PtlMEInsert, PtlMDAttach

Table 3.3 presents a summary of the functions defined by the Portals API. The first column in this table gives the name for the function, the second column gives a brief description of the operation implemented by the function, and the third column identifies the section where the function is defined.

Table 3.3: Functions Defined by the Portals 3.3 API

Name	Operation	Section
PtlACEntry	update an entry in an access control table	3.12
PtlEQAlloc	create an event queue	3.11
PtlEQGet	get the next event from an event queue	3.11
PtlEQFree	release the resources for an event queue	3.11
PtlEQPoll	poll for a new event on multiple event queues	3.11.11
PtlEQWait	wait for a new event in an event queue	3.11
PtlFinl	shutdown the Portals API	3.4
PtlGet	perform a get operation	3.13.4
PtlGetId	get the id for the current process	3.7
PtlGetJid	get the job id for the current process	3.8
PtlGetPut	perform an atomic swap operation	3.13.6
PtlGetRegion	perform a get operation on a memory descriptor region	3.13.5
PtlInit	initialize the Portals API	3.4
PtlMDAttach	create a memory descriptor and attach it to a match entry	3.10
PtlMDBind	create a free-floating memory descriptor	3.10.4
PtlMDUnlink	remove a memory descriptor from a list and release its resources	3.10
PtlMDUpdate	update a memory descriptor	3.10
PtlMEAttach	create a match entry and attach it to a Portal table	3.9
PtlMEAttachAny	create a match entry and attach it to a free Portal table entry	3.9.3
PtlMEInsert	create a match entry and insert it in a list	3.9
PtlMEUnlink	remove a match entry from a list and release its resources	3.9
PtlNIDist	get the distance to another process	3.5
PtlNIFini	shutdown a network interface	3.5
PtlNIHandle	get the network interface handle for an object	3.5
PtlNIInit	initialize a network interface	3.5
PtlNIStatus	read a network interface status register	3.5
PtlPut	perform a put operation	3.13.2
PtlPutRegion	perform a put operation on a memory descriptor region	3.13.3

Table 3.4 summarizes the return codes used by functions defined by the Portals API. All of these constants are integer values. The first column of this table gives the symbolic name for the constant, the second column gives a brief description of the value, and the third column identifies the functions that can return this value.

Table 3.5 summarizes the remaining constant values introduced by the Portals API. The first column in this table presents the symbolic name for the constant, the second column gives a brief description of the value, the third column identifies the type for the value, and the fourth column identifies the sections in which the value is mentioned.

Table 3.4: Function Return Codes for the Portals 3.3 API

Name	Meaning	Functions
PTL_AC_INDEX_INVALID	invalid access control table index	PtlACEntry
PTL_EQ_DROPPED	at least one event has been dropped	PtlEQGet, PtlWait
PTL_EQ_EMPTY	no events available in an event queue	PtlEQGet
PTL_FAIL	error during initialization or cleanup	PtlInit, PtlFini
PTL_MD_ILLEGAL	illegal memory descriptor values	PtlMDAttach, PtlMDBind, PtlMDUpdate
PTL_IFACE_DUP	duplicate initialization of an interface	PtlNIInit
PTL_IFACE_INVALID	initialization of an invalid interface	PtlNIInit
PTL_EQ_INVALID	invalid event queue handle	PtlMDUpdate, PtlEQFree, PtlEQGet
PTL_HANDLE_INVALID	invalid handle	PtlNIHandle
PTL_MD_INVALID	invalid memory descriptor handle	PtlMDUnlink, PtlMDUpdate
PTL_ME_INVALID	invalid match entry handle	PtlMDAttach
PTL_NI_INVALID	invalid network interface handle	PtlNIDist, PtlNIFini, PtlMDBind, PtlEQAlloc
PTL_PID_INVALID	invalid pid	PtlNIInit
PTL_PROCESS_INVALID	invalid process identifier	PtlNIInit, PtlNIDist, PtlMEAttach, PtlMEInsert, PtlACEntry, PtlPut, PtlGet
PTL_PT_INDEX_INVALID	invalid Portal table index	PtlMEAttach
PTL_SR_INDEX_INVALID	invalid status register index	PtlNISStatus
PTL_ME_LIST_TOO_LONG	match entry list too long	PtlMEAttach, PtlMEInsert
PTL_MD_IN_USE	MD has pending operations	PtlMDUnlink
PTL_ME_IN_USE	ME has pending operations	PtlMEUnlink
PTL_NO_INIT	uninitialized API	<i>all</i> , except PtlInit
PTL_NO_SPACE	insufficient memory	PtlNIInit, PtlMDAttach, PtlMDBind, PtlEQAlloc, PtlMEAttach, PtlMEInsert
PTL_MD_NO_UPDATE	no update was performed	PtlMDUpdate
PTL_PT_FULL	Portal table is full	PtlMEAttachAny
PTL_OK	success	<i>all</i>
PTL_SEGV	addressing violation	PtlNIInit, PtlNISStatus, PtlNIDist, PtlNIHandle, PtlMDBind, PtlMDUpdate, PtlEQAlloc, PtlEQGet, PtlEQWait

Table 3.5: Other Constants Defined by the Portals 3.3 API

Name	Meaning	Base type	Intr.	Ref.
PTL_ACK_REQ	request an acknowledgement	ptl_ack_req_t	3.13.2	
PTL_EQ_NONE	a NULL event queue handle	ptl_handle_eq_t	3.2.2	3.10, 3.10.6
PTL_EQ_HANDLER_NONE	a NULL event queue handler function	ptl_eq_handler_t	3.11.5	3.11.6
PTL_EVENT_GET_START	get event start	ptl_event_kind_t	3.11.1	3.13.4
PTL_EVENT_GET_END	get event end	ptl_event_kind_t	3.11.1	3.13.4
PTL_EVENT_PUT_START	put event start	ptl_event_kind_t	3.11.1	3.13.2
PTL_EVENT_PUT_END	put event end	ptl_event_kind_t	3.11.1	3.13.2
PTL_EVENT_GETPUT_START	getput event start	ptl_event_kind_t	3.11.1	3.13.6
PTL_EVENT_GETPUT_END	getput event end	ptl_event_kind_t	3.11.1	3.13.6
PTL_EVENT_REPLY_START	reply event start	ptl_event_kind_t	3.11.1	3.13.4
PTL_EVENT_REPLY_END	reply event end	ptl_event_kind_t	3.11.1	3.13.4
PTL_EVENT_ACK_START	acknowledgement event start	ptl_event_kind_t	3.11.1	3.13.2
PTL_EVENT_ACK_END	acknowledgement event end	ptl_event_kind_t	3.11.1	3.13.2
PTL_EVENT_SEND_START	send event start	ptl_event_kind_t	3.11.1	3.13.2
PTL_EVENT_SEND_END	send event end	ptl_event_kind_t	3.11.1	3.13.2
PTL_EVENT_UNLINK	unlink event	ptl_event_kind_t	3.11.1	3.10.1
PTL_INVALID_HANDLE	invalid handle	ptl_handle_any_t	3.2.2	
PTL_PID_ANY	wildcard for process id fields	ptl_pid_t	3.2.6	3.9.2, 3.12.1
PTL_NID_ANY	wildcard for node id fields	ptl_nid_t	3.2.6	3.9.2, 3.12.1
PTL_UID_ANY	wildcard for user id	ptl_uid_t	3.2.6	3.9.2, 3.12.1
PTL_JID_ANY	wildcard for job id	ptl_jid_t	3.8	3.12.1
PTL_JID_NONE	jid not supported for process	ptl_jid_t	3.8	
PTL_IFACE_DEFAULT	default interface	ptl_interface_t	3.2.5	
PTL_INSERT_AFTER	insert after	ptl_ins_pos_t	3.9.4	
PTL_INSERT_BEFORE	insert before	ptl_ins_pos_t	3.9.4	
PTL_MD_ACK_DISABLE	a flag to disable acknowledgements	int	3.10.1	
PTL_MD_EVENT_START_DISABLE	a flag to disable start events	int	3.10.1	
PTL_MD_EVENT_END_DISABLE	a flag to disable end events	int	3.10.1	
PTL_MD_MANAGE_REMOTE	a flag to enable the use of remote offsets	int	3.10.1	3.13.2, 3.13.3
PTL_MD_OP_GET	a flag to enable get operations	int	3.10.1	
PTL_MD_OP_PUT	a flag to enable put operations	int	3.10.1	
PTL_MD_THRESH_INF	infinite threshold for a memory descriptor	int	3.10.1	
PTL_MD_TRUNCATE	a flag to enable truncation of a request	int	3.10.1	
PTL_MD_IOVEC	a flag to enable scatter/gather memory descriptors	int	3.10.1	
PTL_NO_ACK_REQ	request no acknowledgement	ptl_ack_req_t	3.13.2	
PTL_PT_INDEX_ANY	wildcard for Portal indexes	ptl_pt_index_t	3.12.1	
PTL_RETAIN	disable unlinking	ptl_unlink_t	3.10.3	
PTL_SR_DROP_COUNT	index for the dropped count register	ptl_sr_index_t	3.2.7	3.5.4
PTL_TIME_FOREVER	a flag to indicate unbounded time	ptl_time_t	??	3.11.11
PTL_UNLINK	enable unlinking	ptl_unlink_t	3.10.3	

Chapter 4

The Semantics of Message Transmission

The portals API uses four types of messages: put requests, acknowledgements, get requests, and replies. In this section, we describe the information passed on the wire for each type of message. We also describe how this information is used to process incoming messages.

4.1 Sending Messages

Table 4.1 summarizes the information that is transmitted for a put request. The first column provides a descriptive name for the information, the second column provides the type for this information, the third column identifies the source of the information, and the fourth column provides additional notes. Most information that is transmitted is obtained directly from the *PtlPut* operation. Notice that the handle for the memory descriptor used in the *PtlPut* operation is transmitted even though this value cannot be interpreted by the target. A value of anything other than `PTL_MD_NONE`, is interpreted as a request for an acknowledgement.

Table 4.1: Information Passed in a Put Request

Information	Type	<i>PtlPut</i> arg	Notes
operation	int		indicates a put request
initiator	ptl_process_id_t		local information
user	ptl_uid_t		local information
job id	ptl_jid_t		local information (if supported)
target	ptl_process_id_t	target_id	
portal index	ptl_pt_index_t	pt_index	
cookie	ptl_ac_index_t	ac_index	
match bits	ptl_match_bits_t	match_bits	
offset	ptl_size_t	remote_offset	
memory desc	ptl_handle_md_t	md_handle	no ack if <code>PTL_MD_NONE</code>
length	ptl_size_t	md_handle	length member
data	bytes	md_handle	start and length members

Table 4.2 summarizes the information transmitted in an acknowledgement. Most of the information is simply echoed from the put request. Notice that the initiator and target are obtained directly from the put request, but are swapped in generating the acknowledgement. The only new piece of information in the acknowledgement is the manipulated length which is determined as the put request is satisfied.

Table 4.3 summarizes the information that is transmitted for a get request. Like the information transmitted in a put request, most of the information transmitted in a get request is obtained directly from the *PtlGet* operation. Unlike put requests, get requests do not include the event queue handle. In this case, the reply is generated whenever the

Table 4.2: Information Passed in an Acknowledgement

Information	Type	Put Information	Notes
operation	int		indicates an acknowledgement
initiator	ptl_process_id_t	target	
target	ptl_process_id_t	initiator	
portal index	ptl_pt_index_t	portal index	echo
match bits	ptl_match_bits_t	match bits	echo
offset	ptl_size_t	offset	echo
memory desc	ptl_handle_md_t	memory desc	echo
requested length	ptl_size_t	length	echo
manipulated length	ptl_size_t		obtained from the operation

operation succeeds and the memory descriptor must not be unlinked until the reply is received. As such, there is no advantage to explicitly sending the event queue handle.

Table 4.3: Information Passed in a Get Request

Information	Type	PtlGet argument	Notes
operation	int		indicates a get operation
initiator	ptl_process_id_t		local information
user	ptl_uid_t		local information
job id	ptl_jid_t		local information (if supported)
target	ptl_process_id_t	target_id	
portal index	ptl_pt_index_t	pt_index	
cookie	ptl_ac_index_t	ac_entry	
match bits	ptl_match_bits_t	match_bits	
offset	ptl_size_t	remote_offset	
memory desc	ptl_handle_md_t	md_handle	
length	ptl_size_t	md_handle	length member

Table 4.4 summarizes the information transmitted in a reply. Like an acknowledgement, most of the information is simply echoed from the get request. The initiator and target are obtained directly from the get request, but are swapped in generating the acknowledgement. The only new information in the acknowledgement are the manipulated length and the data, which are determined as the get request is satisfied.

4.2 Receiving Messages

When an incoming message arrives on a network interface, the communication system first checks that the target process identified in the request is a valid process that has initialized the network interface (i.e., that the target process has a valid Portal table). If this test fails, the communication system discards the message and increments the dropped message count for the interface. The remainder of the processing depends on the type of the incoming message. Put and get messages are subject to access control checks and translation (searching a match list), while acknowledgement and reply messages bypass the access control checks and the translation step.

Acknowledgement messages include a handle for the memory descriptor used in the original *PtlPut* or *PtlPutRegion* operation. This memory descriptor will identify the event queue where the event should be recorded. Upon receipt of an acknowledgement, the runtime system only needs to confirm that the memory descriptor and event queue still exist and that there is space for another event. Should any of these conditions fail, the message is simply discarded.

Table 4.4: Information Passed in a Reply

Information	Type	Put Information	Notes
operation	int		indicates an acknowledgement
initiator	ptl_process_id_t	target	
target	ptl_process_id_t	initiator	
portal index	ptl_pt_index_t	portal index	echo
match bits	ptl_match_bits_t	match bits	echo
offset	ptl_size_t	offset	echo
memory desc	ptl_handle_md_t	memory desc	echo
requested length	ptl_size_t	length	echo
manipulated length	ptl_size_t		obtained from the operation
data	bytes		obtained from the operation

and the dropped message count for the interface is incremented. Otherwise, the system builds an acknowledgement event from the information in the acknowledgement message and adds it to the event queue.

Reception of reply messages is also relatively straightforward. Each reply message includes a handle for a memory descriptor. If this descriptor exists, it is used to receive the message. A reply message will be dropped if the memory descriptor identified in the request doesn't exist. In either of this case, the dropped message count for the interface is incremented. These are the only reasons for dropping reply messages. Every memory descriptor accepts and truncates incoming reply messages, eliminating the other potential reasons for rejecting a reply message.

The critical step in processing an incoming put or get request involves mapping the request to a memory descriptor. This step starts by using the Portal index in the incoming request to identify a list of match entries. This list of match entries is searched in order until a match entry is found whose match criteria matches the match bits in the incoming request and whose memory descriptor accepts the request.

Because acknowledge and reply messages are generated in response to requests made by the process receiving these messages, the checks performed by the runtime system for acknowledgements and replies are minimal. In contrast, put and get messages are generated by remote processes and the checks performed for these messages are more extensive. Incoming put or get messages may be rejected because:

- the Portal index supplied in the request is not valid;
- the access control index supplied in the request is not a valid access control entry;
- the access control entry identified by the index does not match the identifier of the requesting process;
- the access control entry identified by the access control entry does not match the Portal index supplied in the request; or
- the match bits supplied in the request do not match any of the match entries with a memory descriptor that accepts the request.

In all cases, if the message is rejected, the incoming message is discarded and the dropped message count for the interface is incremented.

A memory descriptor may reject an incoming request for any of the following reasons:

- the `PTL_MD_PUT` or `PTL_MD_GET` option has not been enabled and the operation is put, get, or swap;
- the length specified in the request is too long for the memory descriptor and the `PTL_MD_TRUNCATE` option has not been enabled.

Acknowledgments

Several people have contributed to the philosophy, design, and implementation of the Portals message passing architecture as it has evolved. We acknowledge the following people for their contributions: Al Audette, Eric Barton, Lee Ann Fisk, David Greenberg, Eric Hoffman, Gabi Istrail, Chu Jong, Mike Levenhagen, Jim Otto, Kevin Pedretti, Mark Sears, Lance Shuler, Mack Stallcup, Jeff VanDyke, Dave van Dresser, Lee Ward, and Stephen Wheat.

Bibliography

- [1] R. Brightwell, D. S. Greenberg, A. B. Maccabe, and R. Riesen. Massively Parallel Computing with Commodity Components. *Parallel Computing*, 26:243–266, February 2000.
- [2] R. Brightwell and L. Shuler. Design and implementation of MPI on Puma portals. In *Proceedings of the Second MPI Developer's Conference*, pages 18–25, July 1996.
- [3] Compaq, Microsoft, and Intel. Virtual Interface Architecture Specification Version 1.0. Technical report, Compaq, Microsoft, and Intel, December 1997.
- [4] Cray Research, Inc. *SHMEM Technical Note for C, SG-2516 2.3*, October 1994.
- [5] M. Lauria, S. Pakin, and A. Chien. Efficient Layering for High Speed Communication: Fast Messages 2.x. In *Proceedings of the IEEE International Symposium on High Performance Distributed Computing*, 1998.
- [6] A. B. Maccabe, K. S. McCurley, R. Riesen, and S. R. Wheat. SUNMOS for the Intel Paragon: A brief user's guide. In *Proceedings of the Intel Supercomputer Users' Group. 1994 Annual North America Users' Conference.*, pages 245–251, June 1994.
- [7] Message Passing Interface Forum. MPI: A Message-Passing Interface standard. *The International Journal of Supercomputer Applications and High Performance Computing*, 8, 1994.
- [8] Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*, July 1997. <http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html>.
- [9] Myricom, Inc. The GM Message Passing System. Technical report, Myricom, Inc., 1997.
- [10] Sandia National Laboratories. *ASCI Red*, 1996. <http://www.sandia.gov/ASCI/TFLOP>.
- [11] L. Shuler, C. Jong, R. Riesen, D. van Dresser, A. B. Maccabe, L. A. Fisk, and T. M. Stallcup. The Puma operating system for massively parallel computers. In *Proceeding of the 1995 Intel Supercomputer User's Group Conference*. Intel Supercomputer User's Group, 1995.
- [12] Task Group of Technical Committee T11. Information Technology - Scheduled Transfer Protocol - Working Draft 2.0. Technical report, Accredited Standards Committee NCITS, July 1998.

Summary of Changes for Version 3.3

1. Section 3.11.11: added PltEQPoll.
2. Section 3.13.3: added PtlPutRegion.
3. Section 3.13.5: added PtlGetRegion.
4. Section 3.10: added `PTL_MD_EVENT_START_DISABLE` and `PTL_MD_EVENT_END_DISABLE` options.
5. Section 3.11.5: added event queue handler capability.
6. Changed naming scheme to be consistent across the entire API.
7. Moved change summaries to the end of the document.

Summary of Changes for Version 3.2

1. Updated version number to 3.2 throughout the document
2. Section 3.7.2: added `PTL_SEGV` to error list for *PtlGetId*.
3. Section 3.9.2: added `PTL_ML_TOO_LONG` to error list for *PtlMEAttach*.
4. Section 3.9.5: removed text referring to a list of associated memory descriptors.
5. Section 3.10.5: added text to describe unlinking a free-floating memory descriptor.
6. Table 3.2: added entry for `ptl_seq_t`.
7. Section 3.10.1:
 - (a) added definition of `max_offset`.
 - (b) added text to clarify `PTL_MD_MANAGE_REMOTE`.
8. Section 3.10.3: modified text for `unlink_op`.
9. Section 3.5.2: added text to clarify multiple calls to *PtlNlInit*.
10. Section 3.10.3: added text to clarify `unlink_nofit`.
11. Section 4.2: removed text indicating that an MD will reject a message if the associated EQ is full.
12. Section 3.10.5: added `PTL_MD_IN_USE` error code and text to indicate that only MDs with no pending operations can be unlinked.
13. Table 3.4: added `PTL_MD_IN_USE` return code.
14. Section 3.11.4: added user id field, MD handle field, and NI specific failure field to the `ptl_event_t` structure.
15. Table 3.2: added `ptl_ni_fail_t`.
16. Section 3.11.4: added `PTL_EVENT_UNLINK` event type.
17. Table 3.3: removed *PtlTransId*.
18. Section 3.9.2, Section 3.9.4, Section 3.13.2: listed allowable constants with relevant fields.
19. Table 3.3: added *PtlMEAttachAny* function.
20. Table 3.4: added `PTL_PT_FULL` return code for *PtlMEAttachAny*.
21. Table 3.5: updated to reflect new event types.
22. Section 3.2.6: added `ptl_nid_t`, `ptl_pid_t`, and `ptl_uid_t`.
23. Section 3.5.1: added `max_iovs` and `max_match_list_length`.

- 24. Section 3.10: changed `max_offset` to `max_size` and added `PTL_MD_IOV` option.
- 25. Added Section 3.8.
- 26. Added Section 3.13.6.
- 27. Got rid of the chapter with obsolete examples.

Summary of Changes for Version 3.1

Thread Issues

The most significant change to the interface from version 3.0 to 3.1 involves the clarification of how the interface interacts with multi-threaded applications. We adopted a generic thread model in which processes define an address space and threads share the address space. Consideration of the API in the light of threads lead to several clarifications throughout the document:

1. Glossary:
 - (a) added a definition for *thread*,
 - (b) reworded the definition for *process*.
2. Section 2: added section 2.4 to describe the multi-threading model used by the Portals API.
3. Section 3.4.1: *PtlInit* must be called at least once and may be called any number of times.
4. Section 3.4.2: *PtlFini* should be called once as the process is terminating and not as each thread terminates.
5. Section 3.7: Portals does not define thread ids.
6. Section 3.5: network interfaces are associated with processes, not threads.
7. Section 3.5.2: *PtlNlInit* must be called at least once and may be called any number of times.
8. Section 3.11.9: *PtlEQGet* returns `PTL_EQ_EMPTY` if a thread is blocked on *PtlEQWait*.
9. Section 3.11.10: waiting threads are awakened in FIFO order.

Two functions, *PtlNlBarrier* and *PtlEQCount* were removed from the API. *PtlNlBarrier* was defined to block the calling process until all of the processes in the application group had invoked *PtlNlBarrier*. We now consider this functionality, along with the concept of groups (see the discussion under “other changes”), to be part of the runtime system, not part of the Portals API. *PtlEQCount* was defined to return the number of events in an event queue. Because external operations may lead to new events being added and other threads may remove events, the value returned by *PtlEQCount* would have to be a hint about the number of events in the event queue.

Handling small, unexpected messages

Another set of changes relates to handling small unexpected messages in MPI. In designing version 3.0, we assumed that each unexpected message would be placed in a unique memory descriptor. To avoid the need to process a long list of memory descriptors, we moved the memory descriptors out of the match list and hung them off of a single match list entry. In this way, large unexpected messages would only encounter a single “short message” match list entry before encountering the “long message” match list entry. Experience with this strategy identified resource management problems with this approach. In particular, a long sequence of very short (or zero length) messages could quickly exhaust the memory descriptors constructed for handling unexpected messages. Our new strategy involves the use

of several very large memory descriptors for small unexpected messages. Consecutive unexpected messages will be written into the first of these memory descriptors until the memory descriptor fills up. When the first of the “small memory” descriptors fills up, it will be unlinked and subsequent short messages will be written into the next “short message” memory descriptor. In this case, a “short message” memory descriptor will be declared full when it does not have sufficient space for the largest small unexpected message.

This led to two significant changes. First, each match list entry now has a single memory descriptor rather than a list of memory descriptors. Second, in addition to exceeding the operation threshold, a memory descriptor can be unlinked when the local offset exceeds a specified value. These changes have led to several changes in this document:

1. Section 2.2:

- (a) removed references to the memory descriptor list,
- (b) changed the portals address translation description to indicate that unlinking a memory descriptor implies unlinking the associated match list entry—match list entries can no longer be unlinked independently from the memory descriptor.

2. Section 3.9.2:

- (a) removed unlink from argument list,
- (b) removed description of `ptl_unlink` type,
- (c) changed wording of the error condition when the Portal table index already has an associated match list.

3. Section 3.9.4: removed unlink from argument list.

4. Section 3.10.1: added `max_offset`.

5. Section 3.10.3:

- (a) added description of `ptl_unlink` type,
- (b) removed reference to memory descriptor lists,
- (c) changed wording of the error condition when match list entry already has an associated memory descriptor,
- (d) changed the description of the `unlink` argument.

6. Section 3.10: removed `PtlMDInsert` operation.

7. Section 3.10.4: removed references to memory descriptor list.

8. Section 3.10.5: removed reference to memory descriptor list.

9. Section 3.15: removed references to `PtlMDInsert`.

10. Section 4: removed reference to memory descriptor list.

11. Revised the MPI example to reflect the changes to the interface.

Several changes have been made to improve the general documentation of the interface.

- 1. Section 3.2.2: documented the special value `PTL_EQ_NONE`.
- 2. Section 3.2.6: documented the special value `PTL_ID_ANY`.
- 3. Section 3.10.4: documented the return value `PTL_INV_EQ`.
- 4. Section 3.10.6: clarified the description of the *PtlMDUpdate* function.
- 5. Introduced a new section to document the implementation defined values.
- 6. Section 3.15: modified Table 3.5 to indicate where each constant is introduced and where it is used.

Other changes

Implementation defined limits (Section 3.5.2)

The earlier version provided implementation defined limits for the maximum number of match entries, the maximum number of memory descriptors, etc. Rather than spanning the entire implementation, these limits are now associated with individual network interfaces.

Added User Ids (Section 3.6)

Group Ids had been used to simplify access control entries. In particular, a process could allow access for all of the processes in a group. User Ids have been introduced to regain this functionality. We use user ids to fill this role.

Removed Group Ids and Rank Ids (Section 3.7)

The earlier version of Portals had two forms for addressing processes: <node id, process id> and <group id, rank id>. A process group was defined as the collection processes created during application launch. Each process in the group was given a unique rank id in the range 0 to $n - 1$ where n was the number of processes in the group. We removed groups because they are better handled in the runtime system.

Match lists (Section 3.9.2)

It is no longer illegal to have an existing match entry when calling `PtlMEAttach`. A position argument was added to the list of arguments supplied to `PtlMEAttach` to specify whether the new match entry is prepended or appended to the existing list. If there is no existing match list, the position argument is ignored.

Unlinking Memory Descriptors (Section 3.10)

Previously, a memory descriptor could be unlinked if the offset exceeded a threshold upon the completion of an operation. In this version, the unlinking is delayed until there is a matching operation which requires more memory than is currently available in the descriptor. In addition to changes in section, this lead to a revision of Figure 2.5.

Split Phase Operations and Events (Section 3.11)

Previously, there were five types of events: `PTL_EVENT_PUT`, `PTL_EVENT_GET`, `PTL_EVENT_REPLY`, `PTL_EVENT_SENT`, and `PTL_EVENT_ACK`. The first four of these reflected the completion of potentially long operations. We have introduced new event types to reflect the fact that long operations have a distinct starting point and a distinct completion point. Moreover, the completion may be successful or unsuccessful.

In addition to providing a mechanism for reporting failure to higher levels of software, this split provides an opportunity for improved ordering semantics. Previously, if one process initiated two operations (e.g., two put operations) on a remote process, these operations were guaranteed to complete in the same order that they were initiated. Now, we only guarantee that the initiation events are delivered in the same order. In particular, the operations do not need to complete in the order that they were initiated.

Well known proces ids (Section 3.5.2)

To support the notion of “well known process ids,” we added a process id argument to the arguments for `PtlINIInit`.